Writing Technician Interface Scripts

BayRS Version 12.00
Site Manager Software Version 6.00

Part No. 117382-A Rev. A
September 1997

Bay Networks

Bay Networks

4401 Great America Parkway
Santa Clara, CA 95054

8 Federal Street
Billerica, MA 01821

Bay Networks, Inc. Software License Agreement

NOTICE: Please carefully read this license agreement before copying or using the accompanying software or installing the hardware unit with pre-enabled software (each of which is referred to as "Software" in this Agreement). BY COPYING OR USING THE SOFTWARE, YOU ACCEPT ALL OF THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT. THE TERMS EXPRESSED IN THIS AGREEMENT ARE THE ONLY TERMS UNDER WHICH BAY NETWORKS WILL PERMIT YOU TO USE THE SOFTWARE. If you do not accept these terms and conditions, return the product, unused and in the original shipping container, within 30 days of purchase to obtain a credit for the full purchase price

1. License Grant. Bay Networks, Inc. ("Bay Networks") grants the end user of the Software ("Licensee") a personal, nonexclusive, nontransferable license: a) to use the Software either on a single computer or, if applicable, on a single authorized device identified by host ID, for which it was originally acquired; b) to copy the Software solely for backup purposes in support of authorized use of the Software; and c) to use and copy the associated user manual solely in support of authorized use of the Software by Licensee. This license applies to the Software only and does not extend to Bay Networks Agent software or other Bay Networks software products. Bay Networks Agent software or other Bay Networks software products are licensed for use under the terms of the applicable Bay Networks, Inc. Software License Agreement that accompanies such software and upon payment by the end user of the applicable license fees for such software.

2. Restrictions on use; reservation of rights. The Software and user manuals are protected under copyright laws. Bay Networks and/or its licensors retain all title and ownership in both the Software and user manuals, including any revisions made by Bay Networks or its licensors. The copyright notice must be reproduced and included with any copy of any portion of the Software or user manuals. Licensee may not modify, translate, decompile, disassemble, use for any competitive analysis, reverse engineer, distribute, or create derivative works from the Software or user manuals or any copy, in whole or in part. Except as expressly provided in this Agreement, Licensee may not copy or transfer the Software or user manuals, in whole or in part. The Software and user manuals embody Bay Networks' and its licensors' confidential and proprietary intellectual property. Licensee shall not sublicense, assign, or otherwise disclose to any third party the Software, or any information about the operation, design, performance, or implementation of the Software and user manuals that is confidential to Bay Networks and its licensors; however, Licensee may grant permission to its consultants, subcontractors, and agents to use the Software at Licensee's facility, provided they have agreed to use the Software only in accordance with the terms of this license.

3. Limited warranty. Bay Networks warrants each item of Software, as delivered by Bay Networks and properly installed and operated on Bay Networks hardware or other equipment it is originally licensed for, to function substantially as described in its accompanying user manual during its warranty period, which begins on the date Software is first shipped to Licensee. If any item of Software fails to so function during its warranty period, as the sole remedy Bay Networks will at its discretion provide a suitable fix, patch, or workaround for the problem that may be included in a future Software release. Bay Networks further warrants to Licensee that the media on which the Software is provided will be free from defects in materials and workmanship under normal use for a period of 90 days from the date Software is first shipped to Licensee. Bay Networks will replace defective media at no charge if it is returned to Bay Networks during the warranty period along with proof of the date of shipment. This warranty does not apply if the media has been damaged as a result of accident, misuse, or abuse. The Licensee assumes all responsibility for selection of the Software to achieve Licensee's intended results and for the installation, use, and results obtained from the Software. Bay Networks does not warrant a) that the functions contained in the software will meet the Licensee's requirements, b) that the Software will operate in the hardware or software combinations that the Licensee may select, c) that the operation of the Software will be uninterrupted or error free, or d) that all defects in the operation of the Software will be corrected. Bay Networks is not obligated to remedy any Software defect that cannot be reproduced with the latest Software release. These warranties do not apply to the Software if it has been (i) altered, except by Bay Networks or in accordance with its instructions; (ii) used in conjunction with another vendor's product, resulting in the defect; or (iii) damaged by improper environment, abuse, misuse, accident, or negligence. THE FOREGOING WARRANTIES AND LIMITATIONS ARE EXCLUSIVE REMEDIES AND ARE IN LIEU OF ALL OTHER WARRANTIES EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Licensee is responsible for the security of

its own data and information and for maintaining adequate procedures apart from the Software to reconstruct lost or altered files, data, or programs.

4. Limitation of liability. IN NO EVENT WILL BAY NETWORKS OR ITS LICENSORS BE LIABLE FOR ANY COST OF SUBSTITUTE PROCUREMENT; SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES; OR ANY DAMAGES RESULTING FROM INACCURATE OR LOST DATA OR LOSS OF USE OR PROFITS ARISING OUT OF OR IN CONNECTION WITH THE PERFORMANCE OF THE SOFTWARE, EVEN IF BAY NETWORKS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL THE LIABILITY OF BAY NETWORKS RELATING TO THE SOFTWARE OR THIS AGREEMENT EXCEED THE PRICE PAID TO BAY NETWORKS FOR THE SOFTWARE LICENSE.

5. Government Licensees. This provision applies to all Software and documentation acquired directly or indirectly by or on behalf of the United States Government. The Software and documentation are commercial products, licensed on the open market at market prices, and were developed entirely at private expense and without the use of any U.S. Government funds. The license to the U.S. Government is granted only with restricted rights, and use, duplication, or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c)(1) of the Commercial Computer Software—Restricted Rights clause of FAR 52.227-19 and the limitations set out in this license for civilian agencies, and subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013, for agencies of the Department of Defense or their successors, whichever is applicable.

6. Use of Software in the European Community. This provision applies to all Software acquired for use within the European Community. If Licensee uses the Software within a country in the European Community, the Software Directive enacted by the Council of European Communities Directive dated 14 May, 1991, will apply to the examination of the Software to facilitate interoperability. Licensee agrees to notify Bay Networks of any such intended examination of the Software and may procure support and assistance from Bay Networks.

7. Term and termination. This license is effective until terminated; however, all of the restrictions with respect to Bay Networks' copyright in the Software and user manuals will cease being effective at the date of expiration of the Bay Networks copyright; those restrictions relating to use and disclosure of Bay Networks' confidential information shall continue in effect. Licensee may terminate this license at any time. The license will automatically terminate if Licensee fails to comply with any of the terms and conditions of the license. Upon termination for any reason, Licensee will immediately destroy or return to Bay Networks the Software, user manuals, and all copies. Bay Networks is not liable to Licensee for damages in any form solely by reason of the termination of this license.

8. Export and Re-export. Licensee agrees not to export, directly or indirectly, the Software or related technical data or information without first obtaining any required export licenses or other governmental approvals. Without limiting the foregoing, Licensee, on behalf of itself and its subsidiaries and affiliates, agrees that it will not, without first obtaining all export licenses and approvals required by the U.S. Government: (i) export, re-export, transfer, or divert any such Software or technical data, or any direct product thereof, to any country to which such exports or re-exports are restricted or embargoed under United States export control laws and regulations, or to any national or resident of such restricted or embargoed countries; or (ii) provide the Software or related technical data or information to any military end user or for any military end use, including the design, development, or production of any chemical, nuclear, or biological weapons.

9. General. If any provision of this Agreement is held to be invalid or unenforceable by a court of competent jurisdiction, the remainder of the provisions of this Agreement shall remain in full force and effect. This Agreement will be governed by the laws of the state of California.

Should you have any questions concerning this Agreement, contact Bay Networks, Inc., 4401 Great America Parkway, P.O. Box 58185, Santa Clara, California 95054-8185.

LICENSEE ACKNOWLEDGES THAT LICENSEE HAS READ THIS AGREEMENT, UNDERSTANDS IT, AND AGREES TO BE BOUND BY ITS TERMS AND CONDITIONS. LICENSEE FURTHER AGREES THAT THIS AGREEMENT IS THE ENTIRE AND EXCLUSIVE AGREEMENT BETWEEN BAY NETWORKS AND LICENSEE, WHICH SUPERSEDES ALL PRIOR ORAL AND WRITTEN AGREEMENTS AND COMMUNICATIONS BETWEEN THE PARTIES PERTAINING TO THE SUBJECT MATTER OF THIS AGREEMENT. NO DIFFERENT OR ADDITIONAL TERMS WILL BE ENFORCEABLE AGAINST BAY NETWORKS UNLESS BAY NETWORKS GIVES ITS EXPRESS WRITTEN CONSENT, INCLUDING AN EXPRESS WAIVER OF THE TERMS OF THIS AGREEMENT.

Contents

Tables

About This Guide

If you are interested in creating and editing Technician Interface scripts, refer to this guide for

- An overview of variables
- Instructions on creating, using, and saving variables, using the Technician Interface script commands
- The function and syntax of each Technician Interface script command

Audience

This manual is for network administrators with the following background:

- Knowledge of the UNIX operating system and the C programming language
- A general understanding of local area and wide area networking fundamentals
- An understanding of the transmission and management protocols used on your network

Before You Begin

Before using this guide, you must load the router software from the release medium. This guide assumes that you are familiar with the Technician Interface. For information on this interface, refer to *Using Technician Interface Software*.

Conventions

angle brackets (< >)	Indicate that you choose the text to enter based on the description inside the brackets. Do not type the brackets when entering the command. Example: if command syntax is **ping** *<ip_address>*, you enter **ping 192.32.10.12**
bold text	Indicates text that you need to enter, command names, and buttons in menu paths. Example: Enter **wfsm &**
	Example: Use the **dinfo** command.
	Example: ATM DXI > Interfaces > **PVCs** identifies the PVCs button in the window that appears when you select the Interfaces option from the ATM DXI menu.
brackets ([ ])	Indicate optional elements. You can choose none, one, or all of the options.
ellipsis points	Horizontal (. . .) and vertical (⋮) ellipsis points indicate omitted information.
italic text	Indicates variable values in command syntax descriptions, new terms, file and directory names, and book titles.
quotation marks (“ ”)	Indicate the title of a chapter or section within a book.
`screen text`	Indicates data that appears on the screen. Example: `Set Bay Networks Trap Monitor Filters`
separator (>)	Separates menu and option names in instructions and internal pin-to-pin wire connections. Example: Protocols > AppleTalk identifies the AppleTalk option in the Protocols menu.
	Example: Pin 7 > 19 > 20
vertical line (|)	Indicates that you enter only one of the parts of the command. The vertical line separates choices. Do not type the vertical line when entering the command. Example: If the command syntax is
	show at routes | **nets**, you enter either **show at routes** or **show at nets**, but not both.

Acronyms

ANSI	American National Standards Institute
ARP	Address Resolution Protocol
ASCII	American Standard Code for Information Interchange
ATM	asynchronous transfer mode
FDDI	Fiber Distributed Data Interface
MAC	media access control
MIB	management information base
OSI	Open Systems Interconnection
OSPF	Open Shortest Path First
RIP	Routing Information Protocol
SNMP	Simple Network Management Protocol

Ordering Bay Networks Publications

To purchase additional copies of this document or other Bay Networks publications, order by part number from Bay Networks Press™ at the following numbers:

- Phone--U.S./Canada: 888-422-9773
- Phone--International: 510-490-4752
- FAX--U.S./Canada and International: 510-498-2609

The Bay Networks Press catalog is available on the World Wide Web at *support.baynetworks.com/Library/GenMisc*. Bay Networks publications are available on the World Wide Web at *support.baynetworks.com/Library/tpubs*.

Bay Networks Customer Service

You can purchase a support contract from your Bay Networks distributor or authorized reseller, or directly from Bay Networks Services. For information about, or to purchase a Bay Networks service contract, either call your local Bay Networks field sales office or one of the following numbers:

Region	Telephone number	Fax number
United States and Canada	800-2LANWAN; then enter Express Routing Code (ERC) 290, when prompted, to purchase or renew a service contract 508-916-8880 (direct)	508-916-3514
Europe	33-4-92-96-69-66	33-4-92-96-69-96
Asia/Pacific	61-2-9927-8888	61-2-9927-8899
Latin America	561-988-7661	561-988-7550

Information about customer service is also available on the World Wide Web at *support.baynetworks.com*.

How to Get Help

If you purchased a service contract for your Bay Networks product from a distributor or authorized reseller, contact the technical support staff for that distributor or reseller for assistance.

If you purchased a Bay Networks service program, call one of the following Bay Networks Technical Solutions Centers:

Technical Solutions Center	Telephone number	Fax number
Billerica, MA	800-2LANWAN	508-916-3514
Santa Clara, CA	800-2LANWAN	408-495-1188
Valbonne, France	33-4-92-96-69-68	33-4-92-96-69-98
Sydney, Australia	61-2-9927-8800	61-2-9927-8811
Tokyo, Japan	81-3-5402-0180	81-3-5402-0173

Chapter 1
Creating a Script File

The Technician Interface script facility allows you to read and execute Technician Interface commands from a script file. You create and edit script files on a remote workstation and transfer the files to the router via TFTP or XMODEM. For instructions on using TFTP or XMODEM, see *Using Technician Interface Software*.

We provide several script programs that let you manage the router using information stored in the management information base (MIB). You can use the scripts to display information about protocols and network services and to enable and disable protocols, circuits, lines, and services. You may also choose to write your own scripts, using this manual as a guide. Technician Interface scripts are very similar to UNIX shell scripts.

This chapter describes how to perform the following tasks, using the Technician Interface script commands:

- Prompting the user for input, using the special variable **$***<prompt>*
- Referencing and using MIB information, using the special variable **$***<object.attribute.instance>*, the **mibget** and **instenv** commands, and formatting MIB entries using the **octetfmt** command
- Creating, using, and deleting variables, using the **arrayenv**, **cutenv**, **enumenv**, **getenv**, **instenv, let**, **setenv**, **sprintf**, and **unsetenv** commands
- Changing a local variable into a global variable, using the **export** command
- Evaluating an arithmetic or logical expression, using the **let** command
- Parsing a text string, using the **cutenv** command
- Controlling the execution of the script, using the **if**, **goto**, **pause**, **gosub**, and **return** commands

- Writing messages to the console, using the **echo** and the **printf** commands
- Recording console messages to a file, using the **record** command
- Handling and recovering from errors, using the **on error** command
- Inserting comments into a script file, using the pound sign (#)
- Debugging a script file, using the **verbose** command
- Saving or restoring variables to or from a file, using the **save env** and **source env** commands
- Running a script file with command line arguments, using the **run** command
- Documenting a script file with command lines

About Variables

A variable is a location in the computer's memory for storing a value. A variable name identifies the location. Variable names can consist of up to 15 alphanumeric characters and the underscore character (_). The first character of a variable name must be an alphabetical character. The Technician Interface stores the value of a variable as a string of ASCII characters. The maximum string length is 255 characters.

You can assign a value to a variable using one of the set variable commands (**arrayenv, cutenv**, **enumenv**, **instenv**, **let, setenv**, **mibget,** or **sprintf**). For instructions on using these commands, see Chapter 2.

After you have assigned a value to a variable, you can refer to the variable on a command line within the script file by entering a dollar sign ($) before the variable name (for example, **$a**).

When you refer to a variable, the Technician Interface substitutes the value set for the variable itself. If you want to prevent or delay the substitution of the variable, enter the backslash character (\) before the $. The backslash character, also called the *escape character*, indicates that special characters follow.

In the following script, the value of **a** is substituted for **$a** when **b** is executed.

```
setenv   a   blue
setenv   b   "My color is \$a"
echo   $b
```

`My color is blue` appears on the user's console.

To prevent a variable from being expanded, enter two dollar signs ($$) before the variable name, as shown in the following example:

```
setenv   a   blue
setenv   b   “My color is \$a”
echo   $$b
```

`My color is $a` appears on the user’s console.

You may embed variables in quoted strings, as shown in the previous example. The following command line encloses the variable names **a** and **b** in braces (**{ }**) to separate the variable names from other data within the string.

```
echo   “The sum of ${a}+1=${b}”
```

Note: If the variable name is followed by a space or an end-of-line character (carriage return), the brackets are optional.

A variable name is usually a literal string of characters preceded by a dollar sign ($). The variable name can also be “built” dynamically when the script is executed by using a combination of literal text and the text stored in other variables.

For example, if you have a variable named *array_10*, you can reference this variable by entering **$array_10** on a command line. You can build this same variable name by concatenating the text strings **array**_ and **10**. For example, if you assign the variable *index* the value 10, using the command **let index = 10**, then you can build the variable name *array_10* dynamically on the command line using the following syntax: **${array_[$index]}**. Notice the use of the brackets (**[ ]**) within the variable name. Any literal text or variable name specified within the brackets is first expanded to replace any variables with their values and then concatenated onto the preceding characters of the variable name. To use brackets, you must enclose the entire variable name being built within braces (**{ }**).

Local, Global, and System Variables

When you first define a variable, the computer stores it in the local environment variable table. *Local variables* are only accessible from within the script in which they were created. You cannot access them from another script file. When a script routine ends, any local variables that you defined are automatically deleted.

Global variables are stored in the global environment variable table and may be accessed by any script file. You create global variables by using the **export** command to change a local variable to a global variable. For more information about the **export** command, see Chapter 2. Global variables are not automatically deleted when a script file ends.

System variables, which are read-only, are created only during a Telnet session.

You cannot have the same variable name in the local and the global variable tables. You can delete local and global variables by using the **unsetenv** command. You can also change the value of a global variable by using any of the set variable commands (**arrayenv, cutenv**, **enumenv**, **instenv**, **let, mibget**, **setenv**, or **sprintf**). For more information about these commands, see Chapter 2.

Note: The value of a variable in the local or global table takes precedence over a variable with the same name in the system variable table.

Special Variables

This section describes how to use input variables, how to prompt for user entry from the console, and how to access and format information from the MIB.

Input Parameters

The Technician Interface reserves the following variables as input parameters for a script file: **$1**, **$2**, **$3**, **$4**, **$5**, **$6**, **$7**, **$8**, **$9** and **$#**. The special variable **$#** contains the number of parameters entered on the command line following the script file name. It is set to zero if no parameters are entered. For more information about input parameters, see "**run**" in Chapter 2.

Prompting for Input

The input prompt variable allows you to create a prompt that accepts user input from the Technician Interface console. The syntax of the input prompt variable is **$***<prompt_string>*. The Technician Interface prompts the user with the message you entered between the brackets (< >). In the following script,

```
setenv   ans   "$<Enter your name: >"
echo   "Your name is $ans"
```

`Enter your name:` appears on the user's console. After the user enters a value, the Technician Interface substitutes the value for **ans** and displays the message `Your name is` followed by the name entered by the user.

For instructions on using the **setenv** command and the **echo** command, see Chapter 2.

Polling the Console for Input

You can specify how much time a user has to respond to a prompt (that is, to press Return) before the system times out. If you do not specify a timeout value, the system prompts the user for input until the user responds.

To specify a timeout value, enter the following command, where *<value_in_seconds>* is the length of time, in seconds, that the system will wait for a response before timing out.

let SYS_IO_TIMEOUT = <value_in_seconds>

The timeout value cannot exceed the timeout specified by one of the following MIB objects: *wfSerialPort.wfSerialPortTimeOut.0* or *wfTelnet.wfTelnetSerialPortTimeOut.0*. For example, if you specify a timeout of 600 seconds and the timeout specified by *wfSerialPort.wfSerialPortTimeOut.0* = 5 minutes, the system uses the timeout value given by *wfSerialPort.wfSerialPortTimeOut.0*.

For instructions on using the **let** command, see Chapter 2.

In the following script, if the user does not press Return before the time specified for SYS_IO_TIMEOUT has elapsed, `nothing entered` appears on the user's console:

```
let   SYS_IO_TIMEOUT   = 10
setenv   ans   "$<Enter your name: >"
echo   "Your name is $ans"
if    "$ans" = "" then;   \
echo "nothing entered"
```

To change the timeout back to an indefinite waiting period, use the following command:

unsetenv SYS_IO_TIMEOUT

For more information about using the commands in the preceding examples, see Chapter 2.

Accessing MIB Information

The Technician Interface script facility allows you to assign the value of a MIB attribute to a variable. Using the syntax **$**<*object.attribute.instance*>, you can refer to the following types of MIB variables:

- Counter variables, which keep track of how many times an event occurs.
- Display string variables, which specify a piece of information stored in ASCII characters, and which the system can display for the user to read. The string must be shorter than 255 characters.
- Gauge variables, which keep track of values that fluctuate.
- Integer variables, which specify information in the form of a simple integer.

The variable values *<object>*, *<attribute>*, and *<instance>* are defined as follows:

<object> is the name or identifier of the object (for example, wfSnmp).

<attribute> is the name or identifier of the attribute (for example, wfSnmpDisable).

<instance> is the identifier of a nontabular object or the index value of a tabular object (for example, 1).

For more information about *<object. attribute. instance>*, see *Using Technician Interface Software*.

Using the syntax **$***<object.attribute.instance [index]>*, you can refer to the following types of data:

- Opaque string data
- Octet string data

Opaque and octet string data are hexadecimal numbers that start with 0x.

The variable value *index* is the byte offset into the opaque or octet string data. An offset of 0 returns the length of the data in bytes. Each byte is accessed, one byte at a time, using the index number. For example, to access the value of a byte in an octet string, use the following procedure:

1. **Enter $<***object.attribute.instance***[0]> (for example, $(wfHwBase.wfHwBpRev.0[0])) to get the length of the data in bytes.**

   The system returns the number of bytes in the octet string (for example, 4).

2. **Enter $<***object.attribute.instance***[***n***]> (for example, $(wfHwBase.wfHwBpRev.0[3])) to get the value of the selected byte (for example, the third byte).**

Formatting a MIB Entry

The **octetfmt** command formats a MIB entry with an Octet or Opaque data type using the specified format type. For more information about using the **octetfmt** command, see Chapter 2.

Defining a Pseudo-Variable Array

The Technician Interface script facility allows you to define a pseudo-variable array. A true *array* is a set of consecutive memory locations used to store data. Each item in the array is called an *element*. An element is a variable. To reference an element of an array, you use a number called the *index*. In a pseudo-variable array, the elements (variables) are not in consecutive memory locations. To create a pseudo-array, use variable names that end in numbers (for example, **a1**, **a2**, and **a3**). The following script shows how to access a member of an array using an index:

```
setenv   a1   one
setenv   a2   two
setenv   a3   three
let   index=2
echo “a[$index]=${a[$index]}”
```

`a[2]=two` is displayed on the user’s console.

For specific instructions on using the commands in the previous example, see Chapter 2.

Creating and Using Variables

This section explains how to view local and global variables, how to set variables, and how to delete them.

Viewing Variables

You view the current list of variables stored in the local and global environment variable tables using the **getenv** command. For example, if you enter

getenv a

the system responds `a = blue`.

You can also use the **echo $***<variable_name>* command to view a variable.

For specific instructions on using the **getenv** and **echo** commands, see Chapter 2.

If you access the Technician Interface via Telnet, the Technician Interface also displays the variables in the system environment variable table. Variables in this table are read-only from the Technician Interface. If your Telnet client supports the *environment variable* option, you can use Telnet to send your UNIX environment variables to the Technician Interface's system list. See your Telnet client documentation for instructions.

Note: If two variables, one in the system table and one in the local or global table, have the same name, the one in the local or global table is used.

Setting Variables

The Technician Interface provides several ways to set variables, depending on what you want to accomplish in your script. This section lists the ways you can set variables. For details on each of the following commands, see Chapter 2.

- To assign an ASCII string or numeric value to a variable in the local environment variable table, use the **setenv** command.

  When you assign a value to a variable, you do not type the **$** before the variable. If the value contains spaces or tabs, place double quotes ("") around the value string. In the following command, you assign the value **Statistics Menu** to the variable **a**.

  setenv a "Statistics Menu"

- To define a pseudo-variable array that contains the list of MIB instance IDs for a given MIB object name, use the **instenv** command.

  The **instenv** command builds a pseudo-variable array by appending an index number to the variable name specified on the command line. Each member of the pseudo-variable array contains a single instance ID. The size of the pseudo-array is stored at index 0.

Note: We recommend that you use the **mibget** command instead of the **instenv** command. The **mibget** command accesses MIB tables approximately 50 percent faster than the **instenv** command.

- To select parts of a text string and write them to a pseudo-variable array, use the **cutenv** command.

- To convert and format text, and save the result in a specified variable for later use, use the **sprintf** command.
- To evaluate a simple arithmetical or logical expression and assign the result to a given variable, use the **let** command.

  The **let** command evaluates an expression from left to right (default). You can use parentheses to change the order of the evaluation.
- To write a list of command line arguments into a pseudo-variable array, use the **arrayenv** command.

  You can append arguments to the end of an existing variable array with the **-a** option. If an argument contains spaces, place double quotes (“”) around it (for example, **“enclose this one”**).
- To assign a sequence of values to a list of variable names, use the **enumenv** command. You must indicate a starting number and, optionally, you can specify a value by which to increment the values in the series. You can use the sequence you have created as named indices of an array variable. Primarily, you use this command with the **arrayenv** command to index the values created by that command.

Deleting Variables

To delete one or more variables from the local or global environment variable table, use the **unsetenv** command. For example, if you enter

unsetenv a

the system deletes variable **a** from the table.

You cannot delete a variable from the system environment variable table.

You can delete all variables that are part of a pseudo-variable array by using the wildcard character *. For example, to delete all variables that begin with **array_**, enter the following command:

unsetenv array_*

For specific instructions on using the **unsetenv** command, see Chapter 2.

Setting the Current Volume or Directory

When you set the current working volume or directory using the **cd** (change directory) command, you are setting the global variable “PWD” to the value you specify in the command, as shown in the following script:

```
cd 3:
echo $PWD
3:
getenv PWD
PWD = “3:”
```

Controlling Program Flow

The Technician Interface provides several commands to control program flow in your script. For details on each of the following commands, see Chapter 2.

- To specify the next line to be executed from the script file, use the **goto** command.

  Within the script file, use the **goto** command with a *label*. A label begins and ends with a colon (:), consists of up to 15 alphanumeric characters and the underscore character (_), and must be on its own line, beginning in column 1.

- To evaluate whether an expression is true, use the **if** command. The expression can compare two numerical values or two ASCII strings. If the expression is true, the script interpreter executes any additional commands that are on the same command line as the **if** command. If the expression is false, the script interpreter does not execute the **if** command.

  You use a semicolon (;) to end an **if** command. The semicolon separates the **if** command from the commands you want executed if the **if** command is true.

- To suspend the Technician Interface’s operation for a given interval, use the **pause** command. During this time the router is still running.

- To call a subroutine inside the same script file, use the **gosub** command. It must be the last command on a line. You can nest subroutines up to 10 deep. You use the **return** command in a subroutine to return to the calling routine. When the **return** command executes, the script interpreter begins executing the instructions on the line following the line containing the **gosub** command.

Writing Messages to the Console

The Technician Interface provides two commands to display messages on the console:

- Use the **echo** command to display command line arguments to the user's terminal.
- Use the **printf** command to convert, format, and print input arguments on the Technician Interface's console.

For details on each of these commands, see Chapter 2.

Saving Console Messages to a File

To save to a file all messages that are written to the console terminal, use the **record** command.

The record file must be opened before recording, and the file must be closed when recording is completed. The file will be lost if it is not closed before the file system is unmounted or the router is reset.

For specific instructions on using the **record** command, see Chapter 2.

Performing Error Recovery

To specify an error handler label within a script file, use the **on error** command. If a command returns an error or if you attempt to abort the script file, the script interpreter goes to the error handler label defined in the file. If you press Ctrl-c to abort the script, the Technician Interface displays the following prompt:

```
Terminate script file? (y/n):
```

You can use the **on error** command with the **-s** flag to allow the script to recover without returning an error message if the **on error** command cannot locate a specified script file. The following example shows the **on error** command with the **-s** flag:

```
on error  -s   :ERROR_HANDLER:
```

For more information about the **on error** command, see Chapter 2 in this manual.

Inserting Comments

You can insert comments into the script file. Comments begin with a pound sign (#) in column 1, as shown in the following example:

```
# The 'echo' command is used to
# output messages to the user's console.
```

When you execute a script file, the system does not display the comments on the user's console.

Debugging a Script File

To enable the debug trace facility, use the **verbose** command. When you enable the trace facility, the Technician Interface displays each command read from the script file before and after its variables have been expanded. For instructions on using the **verbose** command, see Chapter 2.

Saving and Restoring Variables

To save the current list of local and global variables to a file, use the **save env** command. For example, to save the variables to a file named *test*, enter the following command:

save env test

To read and execute the commands in a Technician Interface script file, and to ensure that the router *saves* (restores) all local variables when the script ends, use the **source env** command.

For example, to run the script file *test* on volume 2, and to restore upon completion all local variables saved previously to *test*, enter the following command:

source env 2:test

To locate script errors, use the **verbose on** command and then use the **source env** command. The Technician Interface displays each line from the file before it executes the line, enabling you to easily locate errors.

For more information about the **save env** and **source env** commands, see Chapter 2.

Running a Script File

To read and execute the commands in a Technician Interface script file, and to ensure that the router *deletes* all local variables when the script ends, use the **run** command.

For example, to run the script file *test* on volume 2, deleting upon script completion all local variables saved previously to *test*, enter the following command:

run 2:test

To locate script errors, use the **verbose on** command and then use the **run** command. The Technician Interface displays each line from the file before it executes the line, enabling you to locate errors easily.

You can use the **run** command with the **-s** flag to prevent the system from returning an error message if the **run** command cannot locate the specified script file. The following example shows the **run** command with the **-s** flag:

```
run  -s  install.bat
```

Note: The **run** command automatically appends the *.bat* suffix to the end of the script file name you specify. If you issue the command **run install**, the system first looks for a file named *install*. If it cannot find this file, it looks for a file named *install.bat*.

For more information about the **run** command, see Chapter 2.

Creating Menus

The *menu* utility allows you to edit existing script menus and to create your own menus. For instructions on using the *menu* utility, refer to *Using Technician Interface Scripts*.

You can also edit a script menu by directly editing the appropriate *.mnu* file.

Appendix A shows examples of a script menu and a script.

Chapter 2
Command Reference

In the following pages you can find the function, syntax, and examples of all the script commands you need to write, edit, and execute your script files. Table 2-1 lists the script commands and their functions.

Table 2-1. Script Commands

Command	Function
arrayenv	Writes list of command line arguments to a pseudo-variable array
cutenv	Selects part of a text string and writes it to a pseudo-variable array
echo	Displays command line arguments to a user's terminal
enumenv	Assigns a sequence of values to a list of variables
export	Moves one or more local variables to the global environment variable table
getenv	Displays the current list of variables stored in the local and global environment variable tables
gosub	Calls a subroutine inside the same script file
goto	Specifies the next line to be executed from the script file
if	Evaluates whether an expression is true
instenv	Defines a pseudo-variable array that contains the list of MIB instance IDs for a given MIB object name
let	Evaluates a simple arithmetical or logical expression and assigns the result to a given variable
mibget	Searches a MIB object table one record at a time and retrieves a set of attributes from each record
octetfmt	Formats a MIB entry with an Octet or Opaque data type

(continued)

Table 2-1. Script Commands *(continued)*

Command	Function
on error	Specifies an error handler label within a script file
pause	Suspends the Technician Interface's operation for a given interval
printf	Converts, formats, and prints the input arguments on the Technician Interface's console
record	Saves to a file all messages written to the console terminal
return	Returns control to the calling routine
run	Reads and executes the Technician Interface commands in a script file; deletes local variables when the script ends
save env	Saves the current list of local and global variables to a script file
setenv	Assigns an ASCII string value or a numeric value to a variable in a local environment variable table
source env	Reads and executes the Technician Interface commands in a script file; saves (restores) local variables when the script ends
sprintf	Converts and formats text, and saves the result in a specified variable
unsetenv	Deletes one or more variables from the local or global environment variable table
verbose	Enables the debug trace facility

arrayenv

The **arrayenv** command allows you to write a list of command line arguments *<text_string>* to a pseudo-variable array, beginning with the prefix *<variable_name>*. You can also append arguments to the end of an existing variable array using the **-a** option.

Syntax

The **arrayenv** command has the following syntax:

arrayenv [**-a**] <*variable_name*> **“***<text_string>***” [“***<text_string>***”. . .]**

-a appends the new arguments to the end of a variable array.

<variable_name> is the name of the variable that holds the array.

<text_string> contains the command line arguments. If the text for the arguments contains blanks or special characters, enclose it in double quotes (as shown in syntax). For example, to pass the argument Router One, specify **“Router One”** in the statement.

Example

The following example sets up an array named **attribute**, specifies a series of values for the array, appends a value to the list, and then prints the number of elements in the array, the value of the first element and the value of the last element. The size of the pseudo-variable array is stored at index 0.

```
arrayenv attribute one two “three & four” five “six & seven”
arrayenv -a attribute “Router One”
echo “Size:             ${attribute[0]}”;
echo “First element:    ${attribute[1]}”;
echo “Last element:     ${attribute[${attribute[0]}]}”;
```

The output from this script is as follows:

```
Size: 6
First element: one
Last element: Router One
```

cutenv

The **cutenv** command allows you to select portions of a *<text_string>* (as specified by *<list>*) and write them to a pseudo-variable array, beginning with the prefix *<variable_name>*. The items specified by *<list>* can represent column position or fields delimited by a special character. Column and field numbering start at 1.

Syntax

The **cutenv** command has the following syntax:

cutenv [**-s**] [**-d** *<delimiter>*] [**-f** *<list>* **| -c** *<list>*] *<variable_name>* **"***<text_string>***"**

-s suppresses *<text_string>* with no delimiter characters.

-d *<delimiter>* sets the field delimiter to *<delimiter>*. The default is a tab. Characters with a special meaning to the Technician Interface interpreter, such as a tab or space character, must be set in quotes.

-f *<list>* is field position. Instead of character positions, *<list>* specifies fields that are separated by a delimiter (for example, a tab). Separate variables are created for each noncontiguous field of characters selected. For example:

1,4,7 represent fields 1, 4, and 7.

-c *<list>* is character position. *<list>* is a comma-separated list of integer field numbers (in increasing order), with an optional hyphen (**-**) to indicate ranges. Separate variables are created for each noncontiguous set of characters. For example:

2,4,6 represent characters 2, 4, and 6.

1-5,9 represent characters 1 through 5, inclusive, and 9.

-5,10 represent the first character in the list through 5, inclusive, and 10.

5- represent the character 5 through the last character in the list, inclusive.

Numbers and number ranges may be repeated, overlapping, and in any order. You can select fields or columns not present in the text string.

<variable_name> is the prefix or name representing the pseudo-variable array.

<text_string> with no field delimiters is normally passed through in its entirety.

Example

The following sample script shows how you use the **cutenv** command to assign portions of 192.32.100.55 to a pseudo-variable array:

```
cutenv -d . -f   1,3   addr_  “192.32.100.55”
echo $addr_0 $addr_1 $addr_2
2 192 100
```

echo

The **echo** command allows you to display its command line arguments to the user's terminal.

Syntax

The **echo** command has the following syntax:

echo *<text_string>*

or

echo $*<variable_name>*

<text_string> is any literal text.

<variable_name> is the name of a variable.

Example

For example, when you execute the script

```
echo Good Morning
```

the system displays `Good Morning` on the user's console.

When you execute the script

```
setenv b blue
echo $b
```

the system displays `blue` on the user's console.

enumenv

The **enumenv** command lets you assign a sequence of values to a list of variables. You assign a starting number *<starting_#>* and can optionally specify a value by which to increment the enumeration <**+***increment*>. Most commonly, you use this command with the **arrayenv** command to create named indices to use with a value array. That way you can refer to the elements of the array with meaningful names rather than index numbers. For an example, see the **mibget** command.

Syntax

The **enumenv** command has the following syntax:

enumenv *<starting_#>* **[+***<increment>***]** *<variable_name>* **[***<variable_name>* **. . .]**

<starting_#> is the starting number of the enumeration.

+<increment> is the value by which to increment the enumeration.

<variable_name> is the pseudo-variable array.

Example

The following example creates indices for attributes in a value array. In the result, 1 is the starting number and the values increment by 1. Therefore, the variable named **proto** is 1, the variable named **dest** is 2, and so on.

```
enumenv 1 proto dest next hop mask metric as
```

export

The **export** command allows you to move one or more local variables to the global environment variable table. (When you first define a variable, it is stored in the local environment variable table.) Global variables can be accessed from other script files. You cannot move global variables to the local environment variable table.

Syntax

The **export** command has the following syntax:

export *<variable_name>*

or

export *<variable_name> <variable_name> . . .*

Example

For example:

```
setenv  A   “string”
let  B = 1
export  A  B
```

getenv

The **getenv** command allows you to view the current list of variables stored in the local and global environment variable tables. If you access the Technician Interface via Telnet, the Technician Interface also displays the variables in the system environment variable table. Variables in this table are read-only from the Technician Interface. If your Telnet client supports the *environment variable* option, you can use Telnet to send your UNIX environment variables to the Technician Interface's system list. See your Telnet client documentation for instructions.

Syntax

The **getenv** command has the following syntax:

getenv *<variable_name>*

<variable_name> is the name of the variable whose value you want to display.

To display the current list of variables in all tables, enter the **getenv** command with no arguments.

Note: If two variables, one in the system table and one in the local table, have the same name, the one in the local table is displayed.

gosub

The **gosub** command calls a subroutine inside the same script file. It must be the last command on a line. A subroutine begins and ends with a colon (:). You can nest subroutines up to 10 deep. You use the **return** command in a subroutine to return to the calling routine. When the **return** command executes, the script interpreter begins executing the instructions on the line following the line containing the **gosub** command.

Syntax

The **gosub** command has the following syntax:

gosub :*<subroutine_name>***:**

<subroutine_name> is the name of the subroutine.

Example

The following example shows a script using the **gosub** and **return** commands:

```
:main:
    let arg = 1; gosub :SUBROUTINE:
    echo “Return value: $return”
    goto :EXIT:
:SUBROUTINE:
    let return = ($arg*$arg) + $arg + 1
    return
:EXIT:
```

goto

The **goto** command lets you specify the next line to be executed from the script file. Within the script file, the **goto** command is used with a label. A label begins and ends with a colon (:), consists of up to 15 alphanumeric characters and the underscore character (_), and must be on its own line, beginning in column 1.

Syntax

The **goto** command has the following syntax:

goto :*<label_name>***:**

<label_name> is the name of the label.

Examples

In the following script, you instruct the script file to go to the :LABEL: label.

Note: The following examples only work from a script file; the examples will not work from an interactive Technician Interface console.

```
goto   :LABEL:
echo   "This line is not executed."
:LABEL:
```

The argument to the **goto** command can be a variable whose value is a valid label in the script file. This allows for dynamic branching within the script file.

```
setenv   vector   ":LABEL_2:"
goto   $vector
echo   "This line is skipped."
:LABEL_2:
```

if

The **if** command allows you to evaluate whether an expression is true. The expression can be a comparison of two numerical values or two ASCII strings. If the expression is true, the script interpreter executes any additional commands that are on the same command line as the **if** command. If the expression is false, the script interpreter does not execute the **if** command.

You use a semicolon (;) to end an **if** command. The semicolon separates the **if** command from the additional commands you want executed if the **if** command is true.

Syntax

The **if** command has the following syntax:

if [*<option>*] *<string1> <operand> <string2>* **then;** *<command>* [**;** *<command>* ...]

<option> is one of the following arguments:

-u tests whether *<string1>* is an unsigned number.

-num tests whether *<string1>* is a number.

-ip tests whether *<string1>* is an IP address.

-ipx tests whether *<string1>* is an IPX address.

-ic means ignore case when comparing strings.

-file tests whether *<filename>* is present.

-dir tests whether *<directory_name>* is present.

-vines tests whether *<string1>* is a VINES address.

-date tests whether *<string1>* uses the date format MM/DD/YY.

-time tests whether *<string1>* uses the time format HH:MM:SS.

-tz tests whether *<string1>* uses the GMT offset format [+ | -]HH:MM.

<string1> <string2> are ASCII character strings or numbers.

<operand> is one of the following comparison operations:

(=) equal (!=) not equal

(>) greater than (>=) greater than or equal

(<) less than (<=) less than or equal

then is an optional word used for ease of reading.

<command> is a Technician Interface command.

If both *<string1>* and *<string2>* are numbers, the test expression for the **if** command is a numeric comparison. If *<string1>* and *<string2>* are not numbers, a case-sensitive string comparison is done. You specify a case-insensitive comparison using the **-ic** argument.

The syntax for certain data types may also be tested using the **-u**, **-num**, **-ip**, or **-ipx** argument. If the test on *<string1>* is valid, a 1 is substituted for *<string1>*. If the test on *<string1>* is invalid, a 0 is substituted for *<string1>*.

Example 1

The following example shows a script using the **if** command. The comments explain what is happening in the script.

```
setenv   A   “YES”
setenv   B   1993
#string comparison
if   “$A”   =   “YES”   then ; echo   TRUE ; date
#numeric comparison
if   “$B”  !=   “2001”   then ; echo   FALSE ; date
#test for valid IP address syntax
if   -ip  192.88.22.33 = 1   then ; echo  TRUE
#case-insensitive string comparison
if   -ic   “ABC” = “abc”   then ; echo  TRUE
#test for valid date, time, and timezone offset syntax
if -date “10/10/94” = 1 then; echo “Good”
if -time “10:10:44” = 1 then; echo “Good”
if -tz “-5.00”      = 1 then; echo “Good”
```

The backslash character (\) allows you to continue a command line on the following physical line. When used in this way, the backslash is referred to as a *line continuation character*. The line continuation character makes it easier for you to read the script file. The maximum command line size is still restricted to 255 characters.

Example 2

The following example shows an **if** command using the backslash character:

```
if   $num = 1   then ; \
echo   “The number is 1”;
```

We recommend that you indent the additional commands to make the script file easier to read. Also, if there is a large amount of code following the **if** command, use a **goto** command to branch around the conditional code.

Example 3

The following example shows an **if** command with a **goto** command:

```
if   $num !=1 then ;   goto :IF_10:
echo   “the number is 1”;
:IF_10:
```

Example 4

The following script shows how the **if** command is used to pass parameters into the script file.

```
echo   “Number of parameters entered on run line: $#”
let   index = 1

:LOOP:
if   $index > $# then;   goto :LOOP_END:
echo   “Parameter $index: ${[$index]}”
let   index = $index + 1
goto :LOOP:
:LOOP_END:
```

To test whether a variable is defined, use the **if** command with the following syntax:

if $?<*variable_name*>**=1**

where 1=defined.

Example 5

For example, the following script tests whether variable **a** has been defined:

```
setenv   a   5
if   $?a=1   then; \
echo   “Defined”
```

To test whether a file is present, use the **if** command with the following syntax:

if -file *<filename>***=1**

Example 6

For example, the following script tests whether the *install.bat* file is present:

```
if   -file   install.bat=1   then;\
echo  “File present”
```

To test whether a directory is present, use the **if** command with the following syntax:

if -dir *<directory_name>***=1**

Example 7

For example, the following script tests whether volume 2 is present:

```
if   -dir  2:=1   then;\
echo  “Volume 2 present”
```

instenv

The **instenv** command allows you to define a pseudo-variable array that contains the list of MIB instance IDs for a given MIB object name. The **instenv** command builds a pseudo-variable array by appending an index number to the variable name specified on the command line. Each member of the pseudo-variable array contains a single instance ID. The size of the pseudo-array is stored at index 0.

Syntax

The **instenv** command has the following syntax:

instenv *<array_name> <object>*

<array_name> is the name you assign the pseudo-variable array.

<object> is the name of the MIB object table from which to retrieve the information.

Note: We recommend that you use the new **mibget** command instead of the **instenv** command. The **mibget** command accesses MIB tables approximately 50 percent faster than the **instenv** command.

Example 1

The first four lines in the following example show how you list the instance IDs for *wfIpInterfaceEntry* to the console; the last nine lines show how you use the **instenv** command to assign these instance IDs to a pseudo-variable array.

```
$list   -i   wfIpInterfaceEntry
inst_ids = 192.32.23.5.1
           192.32.7.8.2
           192.32.2.3.3
$instenv   ip_ wfIpInterfaceEntry
$echo   Size: ${ip_[0]}
Size: 3
$echo   “1: ${ip_[1]}”
1: 192.32.23.5.1
$echo   “2: ${ip_[2]}”
2: 192.32.7.8.2
$echo   “3: ${ip_[3]}”
3: 192.32.2.3.3
```

Example 2

The following example is an excerpt from a script file showing how to use the **instenv** command.

```
instenv ip_ wfIpInterfaceEntry
echo “Size of array :{ip_[0]}”
let i = 1
:LOOP:
if $i > ${ip_[0]} then; goto :DONE:
    setenv ipstate wfIpInterfaceEntry.wfIpInterfaceState.${ip_[$i]}
    setenv ipaddr wfIpInterfaceEntry.wfIpInterfaceAddr.${ip_[$i]}
    echo “$i: IP Address: $($ipaddr) State: $($ipstate)”
    let i = $i + 1
    goto :LOOP:
:DONE:
```

let

The **let** command allows you to evaluate a simple arithmetical or logical expression and assign the result to a given variable.

Syntax

The **let** command has the following syntax:

let *<flag>*

<flag> formats the result, as follows:

-s in signed decimal (default)

-u in unsigned decimal

-h in hexadecimal

-date as MM/DD/YY

-days as DD days

-time as HH:MM:SS; Modulo 24 hours

-tz as [+|-]HH:MM; Modulo +/-12:00

The following flags may be used together. Flag combinations other than those listed below are invalid.

-date -time as MM/DD/YY HH:MM:SS

-days -time as DD days HH:MM:SS

Example 1

The flags must immediately precede the variable name that represents the result, as shown in the following example:

```
let   -h   a = 1 + 2
```

Example 2

In the following script, variable name **b** is assigned the sum of variable name **a** plus 1.

```
setenv   a   1
let   b=$a+1
```

The **let** command evaluates an expression from left to right. You can use parentheses to change the order of the evaluation.

Use the arithmetical and logical operators listed in Table 2-2 and Table 2-3 with the **let** command.

Table 2-2. Arithmetical Operators

Operation	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponent	**
Remainder	%

Table 2-3. Logical Operators

Operation	Symbol	Definition
1's complement	~	Bit negation operator that provides the bitwise complement of an integer
AND	&	Arithmetic AND
Inclusive-OR	|	Arithmetic inclusive OR
Exclusive-OR	^	Arithmetic exclusive OR
Logical shift left	<<	Bit-shift to the left, far-right bit set to zero
Logical shift right	>>	Bit-shift to the right, far-left bit set to zero

The **let** command provides the following string functions:

- **strlen( )**
- **strindex( )**
- **strrindex( )**
- **strpbrk( )**
- **strmatch( )**
- *<single_character>*

The arguments for each function can be a literal character string, a set of characters enclosed in double quotes (“”), or the name of an environment variable. When you specify an environment variable, do not precede the name with **$**.

If a function returns an offset within a string, an index of 1 refers to the first character position.

The function **strlen( )** returns the number of characters in the parameter *<string>*.

strlen(*<string>***)**

The function **strindex( )** finds the first occurrence of a string in another string. The function **strrindex( )** finds the last occurrence of a string in another string. Both functions return the offset within *<string>* of *<substring>*’s position, or 0 if <*substring*> is not found.

strindex(*<string>***, *<substring>***)**

strrindex(*<string>***, *<substring>***)**

The function **strpbrk( )** searches *<string>* for the first character that matches any of the characters given in the character string *<set>*. The function returns the offset to the first matching character’s position in *<string>*, or 0 if no character of *<set>* appears in *<string>*.

strpbrk(*<string>, <set>***)**

The function **strmatch(***<string>***, *<pattern>***)** compares *<string>* to *<pattern>*. If *<string>* matches the pattern, the function returns a 1; otherwise it returns a 0. You may use wildcards (***** or **?**) in *<pattern>*. A ***** matches any character (or no character), and a **?** matches any character in that position in the string. A character must exist in that position for **?** to match.

strmatch(*<string>***, *<pattern>***)**

You can also get the ASCII code for a given character (indicated as one character within single quotes, such as ‘c’).

The **let** command also provides the following date and time functions:

- **date( )**
- **time( )**
- **timezone( )**

The function **date( )** used with no arguments returns the current date in seconds. If you enter a date string in the format *MM*/*DD*/*YY* (for example, **date(“10/7/95”)**), the function returns the number of seconds from January 1, 1900 to that date. Dates may range from 1/1/00 (January 1, 1900) to 12/31/99 (December 31, 1999). If you omit the year, the function uses the current year.

The function **time( )** used with no arguments returns the current local time in seconds. If you enter a time string in the format *HH*:*MM*:*SS* (for example, **time(“7:04:32”)**), *HH*:*MM*, or *MM*, the function returns the number of seconds from 12:00 midnight to that time. *HH* must be a value between 0 and 23, inclusive. *MM* must be a value between 0 and 59, inclusive. *SS* must be a value between 0 and 59, inclusive.

The function **timezone( )** used with no arguments returns the current time zone in seconds. If you enter a time-zone string in the format [+|-]*HH*:*MM* (for example, **timezone(“-08:00”)**), the function returns the number of seconds for that time zone from the International Date Line. You enter the time-zone string relative to GMT, which is 43200 seconds offset from the International Date Line. The string must begin with either a plus (+) or a minus (-) sign followed immediately by *HH*:*MM*. *HH* must be a value between 0 and 11, inclusive, and *MM* must be a value between 0 and 59, inclusive.

Note: In the previous examples, the date variables *MM*, *DD*, and *YY* represent the numeric values of month, day, and year, respectively. The time variables *HH*, *MM*, and *SS* represent the numeric values of hours, minutes, and seconds, respectively.

Example 3

The following example shows how to use the **-date**, **-time**, and **-days** flags and the **date( )**, **time( )**, and **timezone( )** functions of the **let** command:

```
let -date -time localtime = date() + time()
echo $localtime
10/11/95 15:24:05

let -date -time london = date(“10/7/95”) + time(“7:04”) + \
                                          timezone() - timezone(“+00:00”)

echo $london
10/07/95 12:04:00

let -date -time newyork = date(“10/7/95”) + time(“7:04”) + \
                                          timezone() - timezone(“-05:00”)

let -date -time la = date(“10/7/95”) + time(“7:04”) + \
                                          timezone() - timezone(“-08:00”)

let -days -time time_left = date(“12/25”)-(date()+time())
echo $time_left
74 days 08:34:58
```

Example4

The following example shows how to use the **let** command to set up a slotmask:

```
let   slot = $<Enter Slot Number>
let   -h   slotmask = 0x80000000 >> ($slot - 1)
echo   “slot #” $slot “slotmask” $slotmask
```

More Examples

Additional examples using the **let** command are shown below:

```
let b = 5 * (10**2);
let capA = 'A'
set env letter “A”
let capA = '$letter'
let len  = strlen(“abcdef”)
setenv string “answer=1”
let len  = strlen(string)
let index = strindex(string, “wer”)
let index = strpbrk(string, “=:;”)
if $index = 0 then; echo “Missing Delimiter”
let index = strmatch(string, “pat”)
if $index = 0 then; echo “No match found”
```

mibget

The **mibget** command lets you search a MIB object table one record at a time and retrieve a set of attributes from each record. For example, you can use this command in a script to retrieve information from a routing table and print the information immediately on the console. When you are searching a larger MIB table, this method is faster and more efficient than using the **instenv** command.

Syntax

The **mibget** command has the following syntax:

mibget [**-n**] [**-p** *<pattern>*] *<object> <attribute_variable_array> <instance_id> <value_variable_array> <next_instance_variable>*

-n retrieves the information from the next record in the MIB table. You can use it in a loop to read the next instance following the identity of the instance *<instance_id>* just read. For example, the current ID is 192.32.0.1 and the next ID is 192.32.0.2. If you enter the **-n** option with the current ID, the **mibget** command returns information for the instance ID 192.32.0.2.

-p *<pattern>* specifies an instance ID pattern string. The script returns only instance IDs that match the pattern. For example, **-p 192.32.**** returns all instances whose address starts with 192.32., such as 192.32.0.0, 192.32.1.1, and so on.

<object> is the name of the MIB object table from which to retrieve the information (for example, *wfIPForwardEntry*).

<attribute_variable_array> is an environmental array variable that contains the list of attributes to be retrieved. To create this array variable, you use the **arrayenv** command. Do not type **$** before this variable name, since the input is passed by reference, not by value. If the attribute is an OCTET string, you can use an optional format specifier (/<*format*>) for each array entry. For information on this format specifier, see the **octetfmt** command. If the attribute is not an OCTET string, the **mibget** command ignores the format specifier.

<instance_id> is the first instance to reference in the MIB table. You can use the * wildcard. If you use the **-n** option, this option identifies the instance ID most recently retrieved.

<value_variable_array> is the name of an environmental array variable that stores the values retrieved. The values are in the same order as listed in the attribute array *<attribute_variable_array>*. Do not type **$** before this variable name, since the input is passed by reference, not by value.

<next_instance_variable> is the name of the variable that stores the instance ID of the record currently read. If the routine reaches the end of the table, it sets this variable to the string "END." In this case, the **mibget** command has not read any record information. Do not type **$** before this variable name, since the input is passed by reference, not by value.

Examples

The following example searches through the IP routing table to retrieve a series of attributes and prints the result on the Technician Interface console. The object from which to retrieve information is *wfIpForwardEntry*. The script uses the **arrayenv** command to create the attribute array variable **attr** that lists the attributes from which to retrieve values. Then it uses the **enumenv** command to establish meaningful names for the indices for **attr**.

The script reads the first record with the **nextflag** turned off.

In the loop (:IP_RT_LOOP:), **nextid** is the *<instance_id>* option. **value** is the name of the value array where the script places the values it retrieves. **nextflag** is set to **-n** to read through the records. The script prints the values as it retrieves them and loops again until it has retrieved the desired attributes from the entire set of records.

The following examples show how to retrieve attribute values from the IP routing table using the **mibget** command. The comments (#) explain what is happening in the script.

```
# Purpose: Retrieve attribute values from the IP routing table.

# Retrieve values from the following attributes:

    arrayenv attr  fIpForwardProto wfIpForwardDest \
                   wfIpForwardNextHop wfIpForwardAge \
                   wfIpForwardMask wfIpForwardMetric1 \
                   wfIpForwardNextHopAS

# Indices for attributes in value array where values are placed:
    enumenv 1 proto dest next age mask metric as
# Read the first record with nextflag turned off.
    setenv nextflag “”
```

```
# First instance ID to read, wildcard.
    setenv nextid “*”


# Begin loop to read entire set of records
 :IP_RT_LOOP:
    mibget  $nextflag  wfIpForwardEntry attr $nextid  value nextid
    if “$nextid” = “END” then; goto :IP_RT_END:

# Set get-next record flag.
    setenv nextflag “-n”
    printf “%-15s %-15s %-5d %8d %8d  %-15s” ${value[$dest]}\
           ${value[$mask]} ${value[$proto]} ${value[$age]} \
           ${value[$metric]} ${value[$next]}
    goto :IP_RT_LOOP:
:IP_RT_END:
```

octetfmt

The **octetfmt** command formats a MIB entry with an Octet or Opaque data type using the specified format type.

Syntax

The **octetfmt** command has the following syntax:

octetfmt *<variable_name> <format_type> <MIB_object>*

<variable_name> is the name of the variable.

<format_type> is one of the following values:

ASCII_STRING returns the data as a printable display string. Any data that is not printable is returned as its hex value in the form *<00>*.

DEC_BYTES formats the data as an unsigned decimal number, with each byte separated by a dot.

DEC_WORDS formats the data as an unsigned decimal number, with a dot separating every 2 bytes.

DEC_LONGS formats the data as an unsigned decimal number, with a dot separating every 4 bytes.

HEX_BYTES formats the data as a hexadecimal string, with each byte separated by a dot.

HEX_WORDS formats the data as a hexadecimal string, with a dot separating every 2 bytes.

HEX_LONGS formats the data as a hexadecimal string, with a dot separating every 4 bytes.

HEX_USTRING or **HEX_STRING** formats the data as a hexadecimal string using uppercase digits for ‘A-F.’

HEX_LSTRING formats the data as a hexadecimal string, using lowercase digits for ‘a-f.’

MAC1_ADDRESS or **MAC_ADDRESS** formats the data as a canonical MAC address.

MAC2_ADDRESS formats the data as a noncanonical MAC address.

SIGNED_INTEGER or **INTEGER** formats the data as a 4-byte signed integer.

UNSIGNED_INTEGER formats the data as a 4-byte unsigned integer.

<MIB_object> represents the variable values *<object.attribute.instance>*.

Example

The following example shows a script using the **octetfmt** command. The command formats the MIB object as a hexadecimal string using uppercase digits.

```
octetfmt   serialno   HEX_USTRING   wfHwBase.wfHwBpSerialNumber.0
echo \“Backplane serial number: 0x$serialno\”
```

on error

The **on error** command allows you to specify an error handler label within a script file. If a command returns an error or if the user attempts to abort the script file, the script interpreter goes to the error handler label defined in the file. If you press Ctrl-c to abort the script, the Technician Interface displays the following prompt:

```
Terminate script file? (y/n):
```

Syntax

The **on error** command has the following syntax:

on error [-s] :*<label>***:**

-s allows the script to recover without returning an error message if the **on error** command cannot locate a specified file.

<label> is the name of the error handler label.

Example

In the following script, the script interpreter goes to the label :ERROR_HANDLER: when an error occurs:

```
on error :ERROR_HANDLER:
    echo   “what
    echo   “this line is skipped due to the above”
    echo   “syntax error. (Missing quote)”
:ERROR_HANDLER:
    echo   “The error handler is called because of the”
    echo   “deliberate syntax error in the previous command.”
```

The error handler is automatically canceled after an error is found or if you issue the **on error** command without a label.

pause

The **pause** command allows you to suspend operation of the Technician Interface for a given interval. During this time the router is still running.

Syntax

The **pause** command has the following syntax:

pause *<no._of_seconds>*

<no._of_seconds> is the length of the pause in the operation of the Technician Interface.

Example

The following command suspends operation of the Technician Interface for 10 seconds:

pause 10

printf

The **printf** command converts, formats, and prints the input arguments (*<p1>,. . .,<pN>*) on the Technician Interface console under the control of *<format>*.

Syntax

The **printf** command has the following syntax:

printf *<format> <p1> <p2> . . . <pN>*

<format> is the string describing what to print.

<p1> <p2>...<pN> represent input arguments.

<format> contains text to be printed and *conversion specifications*. Everything inside the *<format>* string is printed, except for the conversion specifications. When the **printf** command locates a conversion specification, it formats and prints the value of the next input argument that follows the <*format*> string. You must have the same number of input arguments as conversion specifications. The syntax rules for *<format>* are described later in this section.

The **printf** command does not automatically output a newline at the end of each line. You must explicitly output a newline as part of either the *<format>* string or in the argument strings. This feature lets you use multiple **printf** commands to output a single line or you can use a single **printf** command to output multiple lines. The special character \n represents a new line.

Table 2-4 lists the special characters you can use in the format and input arguments.

Table 2-4. Special Characters

Character	Meaning
\a	audible bell
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	tab
\xhh	hex. code
\000	octal code

A summary of the flags and conversion codes follows:

<format> **=** **%***<flag><width>.<precision><conversion_specification>*

<flag> represents one of the following symbols:

- minus sign (**-**) left-justifies data; default is right-justified.
- plus sign (**+**) prints a number.
- space ( ) prints a minus sign in front of a negative number, or prints a space in front of a positive number.
- zero (**0**) pads numeric conversions with zeros.
- pound sign (**#**) specifies an alternate output form if the conversion specifier is **x** or **o**.

<width> is a number that specifies the minimum field width. If *<width>* is specified as an asterisk (*), then use the next argument as *<width>*.

period (.) separates *<width>* and *<precision>*.

<precision> specifies the maximum number of characters to be printed from a string, or the minimum number of digits for an integer. If *<precision>* is an asterisk (*), then use the next argument as *<precision>*.

<conversion_specification> can be one of the following:

- **d** or **i** indicates signed decimal notation.
- **o** indicates unsigned octal notation (no leading zero).
- **x** or **X** indicates unsigned hexadecimal notation (no 0x prefix).
- **u** indicates unsigned decimal.
- **c** indicates a single unsigned character.
- **s** indicates a character string.
- A percent sign (**%**) means to print a %.

If you enter the **printf** command incorrectly, the console returns one of the following error messages:

- `printf: Expected numeric argument; invalid string argument found`

  The **%***<conversion_specification>* expected numeric data in the input argument, which contained a text string (“hello there”) instead of a number (123).

- `printf: Format string has more format codes than there are arguments`

  The <*format*> string has more conversion specifications than there are input arguments (<*p1*>,. . .,<*pN*>).

- `printf: Invalid format code`

  You entered an invalid conversion specification in the format string (**%z**).

Example

The following example shows a sample **printf** command:

```
let   slot = 2
printf   “Slot: %5d\n” $slot
```

record

The **record** command saves to an open file all messages written to the console terminal. Use this command when you need to capture a snapshot of Technician Interface output for analysis by Bay Networks Technical Response Center personnel.

You must open a snapshot file before attempting to capture any Technician Interface output to that file.

You must close the file

- After capturing the desired output from the Technician Interface
- Before unmounting the router file system
- Before resetting the router

If you do not follow these instructions, you can lose the contents of your snapshot file.

Enter the following command to open a file, and to save console messages to that file:

record open [**-fileonly**] [**-pause**] *<vol>:<filename>*

open creates and opens a snapshot file. The system sends command output and messages to both the console terminal and the file.

-fileonly writes messages only to the file, not to the terminal. This option is used only within a script, and is otherwise ignored. This option allows a script to write output to a file. The default is for messages to be written to both the file and the terminal.

-pause immediately places the system in pause mode. You can use this option with the **-fileonly** option.

<vol> is the slot number containing the volume used to store the file.

<filename> is the name of the file used to store output from the router's Technician Interface.

Note: When you designate a snapshot file on an NVFS (flash) file system, remember that only one snapshot file can be open at a time. If you use other Technician Interface commands to write to another file when the snapshot file is open on the same file system, the commands fail.

You can stop saving messages to a snapshot file temporarily by using the **pause** option with the **record** command.

To *change* the pause state, enter

record pause [**on** | **off**]

on disables recording.

off re-enables recording.

To *display* the pause state, enter

record pause

To *close and save* the contents of your designated snapshot file, enter

record close

By checking the global variable RECORD_STATE, you can determine whether

- The router is actively saving messages to your snapshot file (RECORD_STATE = ON)
- The router has temporarily stopped saving messages to your snapshot file (RECORD_STATE = PAUSED)
- The router has unconditionally stopped saving messages to your snapshot file (RECORD_STATE = OFF)

return

You use the **return** command in a subroutine to return to the calling routine. When the **return** command executes, the script interpreter begins executing the instructions on the line following the line containing the **gosub** command.

Syntax

The **return** command has the following syntax:

return

Example

The following example shows a script using the **gosub** and **return** commands:

```
:main:
    let arg = 1; gosub :SUBROUTINE:
    echo “Return value: $return”
    goto :EXIT:
:SUBROUTINE:
    let return = ($arg*$arg) + $arg + 1
    return
:EXIT:
```

run

The **run** command allows you to read and execute the Technician Interface commands in a Technician Interface script file. The **run** command *deletes all local variables* when a script ends.

Syntax

The **run** command has the following syntax:

run [**-s**] *<vol>*:*<filename>* [*<p1>* [*<p2>* [*. . .<p9>*]]]

-s prevents the system from returning an error message if the **run** command cannot locate the specified script file.

<vol> is the volume number of the script you want to execute.

<filename> is the name of the script file you want to execute.

<p1>, *<p2>*, *<p3>*, *<p4>*, *<p5>*, *<p6>*, *<p7>*, *<p8>*, *<p9>* are optional input parameters to the script file. They may be referred to within the script file, as follows:

<p1> -> $1

<p2> -> $2

<p3> -> $3

<p4> -> $4

<p5> -> $5

<p6> -> $6

<p7> -> $7

<p8> -> $8

<p9> -> $9

To locate script errors, use the **verbose on** command and then use the **run** command. The Technician Interface displays each line from the file before it executes the line, enabling you to locate errors easily.

Note: The **run** command automatically appends the *.bat* suffix to the end of the script file name you specify. If you issue the command **run install**, the system first looks for a file named *install*. If it cannot find this file, it looks for a file named *install.bat*.

The special variable **$#** contains the number of parameters entered on the command line following the script file name. It is set to zero if no parameters are entered.

save env

The **save env** command allows you to save the current list of local and global variables to a script file.

Syntax

The **save env** command has the following syntax:

save env *<filename>*

<filename> is the name of the file in which you want to store the variables.

Example

For example, the following script saves the current list of variables to the *install.bat* file.

```
save env install.bat
```

setenv

The **setenv** command allows you to assign an ASCII string value or a numeric value to a variable in the local environment variable table. When you assign a value to a variable, you do not type the **$** before the variable. If the value contains spaces or tabs, place double quotes (“”) around the value string.

Syntax

The **setenv** command has the following syntax:

setenv *<variable_name> <variable_value>*

<variable_name> is the name of the variable.

<variable_value> is the value assigned to the variable.

Example

For example, the following script assigns the value “Good Morning” to the variable **a.**

```
setenv  a  “Good Morning”
```

source env

The **source env** command allows you to read and execute the Technician Interface commands in a Technician Interface script file. The **source env** command *saves (restores) all local variables* when a script ends.

Syntax

The **source env** command has the following syntax:

source env *<vol>*:*<filename>*

<vol> is the volume storing the script file.

<filename> is the name of the script file that contains the variables.

To locate script errors, use the **verbose on** command and then use the **source env** command. The Technician Interface displays each line from the file before it executes the line, enabling you to easily locate errors.

sprintf

The **sprintf** command converts and formats text, and saves the result in a specified variable for later use.

Syntax

The **sprintf** command has the following syntax:

sprintf *<variable_name> <format> <p1> <p2> . . . <pN>*

<variable_name> is the name of the environment variable where the formatted string is saved.

<format> is the conversion specification. (See “printf” for information about conversion specifications.)

<p2> . . . <pN> is the list of arguments to be formatted.

If you enter the **sprintf** command incorrectly, the console returns one of the following error messages:

- `sprintf: Expected numeric argument; invalid string argument found`

  The **%***<conversion_specification>* expected numeric data in the input argument, which contained a text string (“hello there”) instead of a number (123).

- `sprintf: Format string has more format codes than there are arguments`

  The <*format*> string has more conversion specifications than there are input arguments (<*p1*>,. . .,<*pN*>).

- `sprintf: Invalid format code`

  You entered an invalid conversion specification in the format string (**%z**).

- `*ERROR*** Symbol name is greater than 15 characters`

  You specified a variable name longer than 15 characters.

- `*ERROR*** Invalid character “<invalid_character>” in <symbol>`

  You entered an invalid character in the variable name.

- `**ERROR*** Symbol text is greater than 255 characters`

  The value assigned to the variable is longer than 255 characters.

unsetenv

The **unsetenv** command allows you to delete one or more variables from the local or global environment variable table. You cannot delete a variable from the system environment variable table.

Syntax

The **unsetenv** command has the following syntax:

unsetenv *<variable_name>*

unsetenv *<variable_name>,<variable_name>, . . .*

unsetenv *****

unsetenv **-l ***

unsetenv **-g ***

unsetenv *<text>******

unsetenv *<text>* ?

<variable_name> is the name of the variable you want to delete.

-l deletes variables from the local table only.

-g deletes variables from the global table only.

***** deletes all variables from the table you specify. If you do not specify a table, it deletes all variables.

? matches a single character in the given string position.

<text> deletes all variables that begin with the specified text string.

Examples

If you enter:	The system:
unsetenv slot	Deletes the variable **slot** from either the local or the global table, depending on the table in which the variable resides.
unsetenv slot value	Deletes the variables **slot** and **value** from either the local or global table, depending on the table in which the variables reside.
unsetenv *	Deletes all variables in both the local and global environment variable tables.
unsetenv -l *	Deletes all variables in the local environment variable table.
unsetenv -g *	Deletes all variables in the global environment variable table.
unsetenv ip_*	Deletes all symbols that begin with **ip**_.

verbose

The **verbose** command allows you to enable the debug trace facility. When you enable the trace facility, the Technician Interface displays each command read from the script file before and after its variables have been expanded.

Syntax

The **verbose** command has the following syntax:

verbose [**on**|**off**]

[**on**|**off**] is **on** to enable the debug trace facility or **off** to disable it.

Appendix A
Sample Scripts

To help you create your own scripts, this appendix contains three sample scripts:

- Menu script
- FDDI.mnu script
- FDDI.bat script

Menu Script

The following example shows a menu script for a Bay Networks router:

```
:MENU:
on error :MENU:
echo “”
echo “Simple Menu Example”
echo “”
echo “1. Display Time and Date”
echo “2. Display TFTP default volume”
echo “3. Set TFTP default volume”
echo “4. Quit”
echo “”
setenv ans “$<Enter selection: >”
if “$ans ” = “” then; \
goto :MENU:
if $ans = 1 then; \
date ; \
echo “” ; \
goto :MENU:
if $ans = 2 then ;\
let vol=$(wfTftp.wfTftpDefaultVolume.0);\
echo “Tftp Default Volume:$vol”;\
goto :MENU:
if $ans = 3 then ;\
```

```
let vol=$<Enter new TFTP Default Value #: >;\
set wfTftp.wfTftpDefaultVolume.0 $vol;\
go to :MENU:
if $ans = 4 then;\
echo “Exiting ...”;\
goto :EXIT:
echo “Bad choice, try again ...”
pause 2
goto :MENU:
:EXIT:
```

If you place a label at the end of the script file, you can terminate the script file at any time by executing a **goto :EXIT:** command.

FDDI.MNU Script

The following example shows a menu script for a Bay Networks router.

```
# @(#)WSCCS j/scripts-fddi.mnu 1.1 4/8/95 #
######################################################################
#  Copyright 1995 Bay Networks, Inc.
######################################################################
    alias setcmd “let cmd_0 = \$cmd_0 + 1; setenv cmd_\$cmd_0 %1;
    let title_0 = \$title_0 + 1; setenv title_\$title_0 %2;”

    setenv maintitle “FDDI Menu”
    setenv cmd_0   “0”
    setenv title_0 “0”

    setcmd “show fddi alerts”     “Alerts”
    setcmd “show fddi base”       “Base Info”
    setcmd “show fddi disabled”   “Disabled Circuits”
    setcmd “show fddi enabled”     “Enabled Circuits”
    setcmd “show fddi version”    “FDDI.bat version”
    setcmd “show fddi hwfilters”  “Hardware Filters”
    setcmd “show fddi mac”        “MAC Parameters”
    setcmd “show fddi port”       “Port Parameters”
    setcmd “show fddi receive errors” “Receive Errors”
    setcmd “show fddi sat”        “SMT Parameters”
    setcmd “show fddi stats”      “Stats”
    setcmd “show fddi system   errors”  “System Errors”
    setcmd “show fddi transmit errors”  “Transmit Errors”

    unalias setcmd
```

FDDI.BAT Script

The following example shows a protocol script for a Bay Networks router. You view output from the script by entering script commands (for example, **show fddi**) at the Technician Interface command line, or by using the menu utility. See *Using Technician Interface Scripts* for instructions on entering script commands.

The following script allows you to display information from FDDI MIB objects, and to enable or disable FDDI.

```
# @(#)WSCCS k/scripts-fddi.bat 1.11 10/31/95 #
######################################################################
#
#   Copyright Bay Networks, Inc. 1995
#
#   Display FDDI Driver Information
#
######################################################################

######################################################################
#
# Set up for either using the builtin octetfmt function (7.80 or newer)
# or the gosub (7.70)
#
######################################################################
#
# General variable initialization...

setenv match_entries 0
setenv matchcct      ""
```

```
#####################################################################
#
# Sub-command vectoring...
#
#####################################################################

if $# = 0   then ; goto :FI_HELP:

setenv cmd “$1”
let cmdlen = strlen(cmd)
if $cmdlen < 3 then ; goto :FI_HELP:

let i = strindex(“show”, cmd)
if $i = 1 then ; goto :FI_SHOW:

let i = strindex(“enable”, cmd)
if $i = 1 then ; goto :FI_ENBL:

let i = strindex(“disable”, cmd)
if $i = 1 then ; goto :FI_DSBL:

# else fall through to the Help screen.


#####################################################################
#
#  Help screen...
#
#####################################################################

:FI_HELP:
    echo “FDDI Command Script”
    echo “--------------------”
    echo
    echo “This script is not intended to be run directly.”
    echo “Please 'run setpath' and then use the following”
    echo “commands:”
    echo
    echo “disable fddi”
    echo “enable  fddi”
    echo “show    fddi”
    echo
    goto :FI_END:
```

```
#####################################################################
#
#  Parse for “show” command...
#
#####################################################################

:FI_SHOW:
if $# = 1 then ; goto :FI_HELP_SHOW:
setenv cmd “$2”
let len = strlen(cmd)
if $len < 3 then ; goto :FI_HELP_SHOW:

let i = strindex(“base”, cmd)
if $i = 1 then ; goto :FI_BASE:

let i = strindex(“alerts”, cmd)
if $i = 1 then ; goto :FI_ALERTS:

let i = strindex(“disabled”, cmd)
if $i = 1 then ; goto :FI_DSBLED:

let i = strindex(“enabled”, cmd)
if $i = 1 then ; goto :FI_ENBLED:

let i = strindex(“stats”, cmd)
if $i = 1 then ; goto :FI_STATS:

let i = strindex(“receive”, cmd)
if $i = 1 then ; goto :FI_RCV:

let i = strindex(“transmit”, cmd)
if $i = 1 then ; goto :FI_XMIT:

let i = strindex(“system”, cmd)
if $i = 1 then ; goto :FI_SYSTEM:

let i = strindex(“smt”, cmd)
if $i = 1 then ; goto :FI_SMT:

let i = strindex(“mac”, cmd)
if $i = 1 then ; goto :FI_MAC:

let i = strindex(“port”, cmd)
if $i = 1 then ; goto :FI_PORT:

let i = strindex(“sample”, cmd)
if $i = 1 then ; goto :FI_SAMP:
```

```
let i = strindex(“hwfilters”, cmd)
if $i = 1 then ; goto :FI_HWF:

let i = strindex(“version”, cmd)
if $i = 1 then ; goto :FI_VER:

:FI_HELP_SHOW:
echo “FDDI Show Command Help”
echo “----------------------”
echo
echo “show fddi <option>”
echo “   Where option is one of the following:”
echo “   ?”
echo “   alerts”
echo “   base            [circuit <circuit name>]”
echo “   disabled”
echo “   enabled”
echo “   hwfilters”
echo “   mac             [circuit <circuit name>]”
echo “   port”
echo “   receive  errors [circuit <circuit name>]”
echo “   sample   [<period in seconds>] [circuit <circuit name>]”
echo “   smt             [circuit <circuit name>]”
echo “   stats           [circuit <circuit name>]”
echo “   system   errors [circuit <circuit name>]”
echo “   transmit errors [circuit <circuit name>]”
echo “   version”
goto :FI_END:

#####################################################################
#
# Base or Module screen...
#
#####################################################################
:FI_BASE:

#
# check for circuit name to match on...
#
    if $# = 2 then ; goto :FI_BASE_NM:
    setenv cmd “$3”
    let len = strlen(cmd)
    if $len < 3 then ; goto :FI_HELP_SHOW:
    let j = strindex(“circuit”, cmd)
    if $j != 1 then ; goto :FI_HELP_SHOW:
    if $# != 4 then ; goto :FI_HELP_SHOW:
    setenv matchcct “$4”
```

```
:FI_BASE_NM:

    setenv nextflag “”
    setenv pattern  “”
    setenv nextid   “*”

    array    attr wfFDDICct wfFDDIState wfFDDIEnable wfFDDISlot
    array -a attr wfFDDINode wfFDDIMadr/MAC_ADDRESS wfFDDIBofl
    array -a attr wfFDDIBoflTmo wfFDDIMtu wfFDDIHardwareFilter

    enum  1   Cct State Enable Slot i
    enum  $i  Node Madr Bofl i
    enum  $i  BoflTmo Mtu HwFilter

    echo
    echo “FDDI Modules:”
    echo “-------------”

    gosub :FI_BASEHDR:

    let i = 0

:FI_LOOP:
    mibget -n wfFddiEntry attr $nextid  value nextid
    if “$nextid” = “END” then; let i = $i + 1; goto :FOOTER_AND_END:

    let cctnum = ${value[$Cct]}
    gosub :GETCCTNAME:

    if “$matchcct” = “” then ; goto :FI_PRNTB:
    if -ic “$matchcct” != “$cctname” then ; goto :FI_BASE_NXT:
    let match_entries = $match_entries + 1

:FI_PRNTB:
    gosub :FI_ONELINE:
#
# Do next circuit...
#
:FI_BASE_NXT:
    let i = $i + 1
    goto :FI_LOOP:
```

```
#####################################################################
#
# Show Alerts screen...
#
#####################################################################
:FI_ALERTS:
    setenv matchcct “xxx”

    setenv nextflag “”
    setenv pattern  “”
    setenv nextid   “*”

    array    attr wfFDDICct wfFDDIState wfFDDIEnable wfFDDISlot
    array -a attr wfFDDINode wfFDDIMadr/MAC_ADDRESS wfFDDIBofl
    array -a attr wfFDDIBoflTmo wfFDDIMtu wfFDDIHardwareFilter

    enum  1   Cct State Enable Slot i
    enum  $i  Node Madr Bofl i
    enum  $i  BoflTmo Mtu HwFilter

    echo
    echo “FDDI Modules on Alert:”
    echo “----------------------”

    gosub :FI_BASEHDR:

    let i = 0

:FI_ALOOP:
    mibget -n wfFddiEntry attr $nextid  value nextid
    if “$nextid” = “END” then; let i = $i + 1; goto :FOOTER_AND_END:

    let cctnum = ${value[$Cct]}
    gosub :GETCCTNAME:

    if ${value[$Enable]} != 1 then ; goto :FI_SKIPA:
    if ${value[$State]}  != 1 then ; goto :FI_BASE10:
    if $(wfFddiSmtEntry.wfFddiSmtCfState.$nextid) = 1 then; \
        goto :FI_BASE10:
    if $(wfFddiMacExtEntry.wfFddiMacMaUnitDataEnable.$nextid) != 2 \
        then; goto :FI_SKIPA:

:FI_BASE10:
    let match_entries = $match_entries + 1
    gosub :FI_ONELINE:
```

```
# Do next circuit...
#
:FI_SKIPA:
    let i = $i + 1
    goto :FI_ALOOP:

########################################################################
#
# Show Disabled screen...
#
########################################################################

:FI_DSBLED:
    setenv matchcct “xxx”

    setenv nextflag “”
    setenv pattern  “”
    setenv nextid   “*”

    array    attr wfFDDICct wfFDDIState wfFDDIEnable wfFDDISlot
    array -a attr wfFDDINode wfFDDIMadr/MAC_ADDRESS wfFDDIBofl
    array -a attr wfFDDIBoflTmo wfFDDIMtu wfFDDIHardwareFilter

    enum  1   Cct State Enable Slot i
    enum  $i  Node Madr Bofl i
    enum  $i  BoflTmo Mtu HwFilter

    echo
    echo “FDDI Modules Disabled”
    echo “---------------------”

    gosub :FI_BASEHDR:

    let i = 0

:FI_DLOOP:
     mibget -n wfFddiEntry attr $nextid  value nextid
    if “$nextid” = “END” then; let i = $i + 1; goto :FOOTER_AND_END:

    let cctnum = ${value[$Cct]}
    gosub :GETCCTNAME:

    if ${value[$Enable]} == 2 then ; \
        let match_entries = $match_entries + 1 ; \
        gosub :FI_ONELINE:

    let i = $i + 1
    goto :FI_DLOOP:
```

```
#####################################################################
#
# Show Enabled screen...
#
#####################################################################

:FI_ENBLED:
    setenv matchcct “xxx”

    setenv nextflag “”
    setenv pattern  “”
    setenv nextid   “*”

    array    attr wfFDDICct wfFDDIState wfFDDIEnable wfFDDISlot
    array -a attr wfFDDINode wfFDDIMadr/MAC_ADDRESS wfFDDIBofl
    array -a attr wfFDDIBoflTmo wfFDDIMtu wfFDDIHardwareFilter

    enum  1   Cct State Enable Slot i
    enum  $i  Node Madr Bofl i
    enum  $i  BoflTmo Mtu HwFilter


    echo
    echo “FDDI Modules Enabled”
    echo “--------------------”

    gosub :FI_BASEHDR:

    let i = 0

:FI_ELOOP:
    mibget -n wfFddiEntry attr $nextid  value nextid
    if “$nextid” = “END” then; let i = $i + 1; goto :FOOTER_AND_END:

    let cctnum = ${value[$Cct]}
    gosub :GETCCTNAME:

    if ${value[$Enable]} == 1 then ; \
        let match_entries = $match_entries + 1; \
        gosub :FI_ONELINE:

    let i = $i + 1
    goto :FI_ELOOP:
```

```
#####################################################################
#
# Print header for base/alerts/disabled/enabled screens...
#
#####################################################################

:FI_BASEHDR:
    echo
    printf "%-4.4s %-4.4s %-8.8s %-8.8s %-17.17s %-4.4s %-4.4s \
           %-8.8s\n" "" "" "" "" ""  "BOFL"  ""  "HW"
    printf "%-4.4s %-4.4s %-8.8s %-8.8s %-17.17s %-4.4s %-4.4s \
           %-8.8s\n" "Slot" "Conn" "Circuit" "State" "MAC Address" \
           "TMO" "MTU" "Filter"
    printf "%-4.4s %-4.4s %-8.8s %-8.8s %-17.17s %-4.4s %-4.4s  \
           %-8.8s\n" "----" "----" "--------" "--------" \
           "-----------------" "----" "----" "--------"
    return

#####################################################################
#
# Print one line for base/alerts/disabled/enabled screens...
#
#####################################################################

:FI_ONELINE:
    printf "%4d %4d" ${value[$Slot]} \
    ${value[$Node]}

    printf "%-8.8s" "$cctname"

    if ${value[$Enable]} = 2 then; \
        printf "%-8s" "Disabled" ; goto :FI_STATE_ESC:
    if ${value[$State]} = 4 then ; printf "%-8s" "Not Pres" ; goto
:FI_STATE_ESC:
    if ${value[$State]} = 3 then ; printf "%-8s" "Init" ; goto
:FI_STATE_ESC:
    if ${value[$State]} = 2 then ; printf "%-8s" "Down" ; goto
:FI_STATE_ESC:
    if $(wfFddiSmtEntry.wfFddiSmtCfState.$nextid) = 1 then; \
        printf "%-8s" "Down" ; goto :FI_STATE_ESC:
    if $(wfFddiMacExtEntry.wfFddiMacMaUnitDataEnable.$nextid) = 2 \
        then; printf "%-8s" "LLC Off" ; goto :FI_STATE_ESC:
    printf "%-8s" "Up"
```

```
:FI_STATE_ESC:
    printf “%-17.17s” “${value[$Madr]}”

    if ${value[$Bofl]} = 2 then ; \
        printf “%-4s” “Off” ; goto :FI_BOFL_ESC:
    printf “%4d” ${value[$BoflTmo]}

:FI_BOFL_ESC:
    printf “%4d”  ${value[$Mtu]}

    let mode = ${value[$HwFilter]}
    if $mode = 1 then ; printf “%-8s” “Enabled”
    if $mode = 2 then ; printf “%-8s” “Disabled”

    echo

    cutenv -c9-15 name2  “$cctname”
    if ${name2[0]} = 1 then; \
        printf “%4s %4s *%-7.7s\n”  “”  “”  “${name2[1]}”

    return

#####################################################################
#
# Statistics screen...
#
#####################################################################
:FI_STATS:

#
# check for circuit name to match on...
#
    if $# = 2 then ; goto :FI_STAT_NM:
    setenv cmd “$3”
    let len = strlen(cmd)
    if $len < 3 then ; goto :FI_HELP_SHOW:
    let j = strindex(“circuit”, cmd)
    if $j != 1 then ; goto :FI_HELP_SHOW:
    if $# != 4 then ; goto :FI_HELP_SHOW:
    setenv matchcct “$4”
```

```
:FI_STAT_NM:

setenv nextflag “”
setenv pattern  “”
setenv nextid   “*”

array    attr wfFDDICct wfFDDICrcErrRx wfFDDIOverrunRx
array -a attr wfFDDIInvalidFrameStatusRx wfFDDIMacErrRx
array -a attr wfFDDIRxOversizedFrames wfFDDIAbortTx wfFDDIUnderrunTx
array -a attr wfFDDIParityErrRx wfFDDIParityErrTx wfFDDIRingErrRx
array -a attr wfFDDIRingErrTx wfFDDISmtRingErrRx wfFDDIPortOpErr
array -a attr wfFDDIInternOpErr wfFDDIHostErr
array -a attr wfFDDISlot wfFDDINode wfFDDIOctetsRxOk wfFDDIFramesRxOk
array -a attr wfFDDIOctetsTxOk wfFDDIFramesTxOk

enum   1 Cct CrcErrRx OverrunRx i
enum  $i InvalidFrame MacErrRx i
enum  $i RxOversized AbortTx UnderrunTx i
enum  $i ParityErrRx ParityErrTx RingErrRx i
enum  $i RingErrTx SmtRingErrRx PortOpErr i
enum  $i InternOpErr HostErr i
enum  $i Slot Node OctetsRxOk FramesRxOk i
enum  $i OctetsTxOk FramesTxOk

echo
echo “FDDI Module I/O Statistics:”
echo “---------------------------”
echo

printf “%-4.4s %-4.4s %-8.8s %-10.10s %-10.10s %-10.10s %-10.10s \
       %-10.10s\n” “”  “”  “”  “Receive”  “Receive”  “Transmit” \
       “Transmit” “Total”
printf “%-4.4s %-4.4s %-8.8s %-10.10s %-10.10s %-10.10s %-10.10s \
       %-10.10s\n” “Slot”  “Conn”  “Circuit”  “Bytes”  “Frames” \
       “Bytes” “Frames”  “Errors”
printf “%-4.4s %-4.4s %-8.8s %-10.10s %-10.10s %-10.10s %-10.10s \
       %-10.10s\n” “----” “----” “--------” “----------” \
       “----------” “----------” “----------” “----------”

    let i = 0
```

```
:FI_STLOOP:
    mibget -n wfFddiEntry attr $nextid  value nextid
    if "$nextid" = "END" then; let i = $i + 1; goto :FOOTER_AND_END:

    let i = $i + 1

    let cctnum = ${value[$Cct]}
    gosub :GETCCTNAME:

    if "$matchcct" = "" then ; goto :FI_PRNTS:
    if -ic "$matchcct" != "$cctname" then ; goto :FI_STAT_NXT:
    let match_entries = $match_entries + 1

#
# Sum up total errors for display...
#
:FI_PRNTS:
    let tlerrs = ${value[$CrcErrRx]} + ${value[$OverrunRx]} + \
    ${value[$InvalidFrame]} + ${value[$MacErrRx]}

    gosub :MAC_EXTRACT:
    let tlerrs = $tlerrs + $mac_sum

    let tlerrs = $tlerrs + ${value[$RxOversized]} + ${value[$AbortTx]}
    let tlerrs = $tlerrs + ${value[$UnderrunTx]}  + \
                  ${value[$ParityErrRx]}
    let tlerrs = $tlerrs + ${value[$ParityErrTx]} + ${value[$RingErrRx]}
    let tlerrs = $tlerrs + ${value[$RingErrTx]}   + \
                  ${value[$SmtRingErrRx]}
    let tlerrs = $tlerrs + ${value[$PortOpErr]}   + \
    ${value[$InternOpErr]}
    let tlerrs = $tlerrs + ${value[$HostErr]}
#
# Display the line...
#
    printf "%4d %4d %-8.8s" ${value[$Slot]} ${value[$Node]}  "$cctname"
    printf "%10u %10u %10u %10u %10u" ${value[$OctetsRxOk]} \
           ${value[$FramesRxOk]} ${value[$OctetsTxOk]} \
           ${value[$FramesTxOk]} $tlerrs

    echo
    cutenv -c9-15 name2  "$cctname"
    if ${name2[0]} = 1 then; \
       printf "%4s %4s *%-7.7s\n" ""  ""  "${name2[1]}"

:FI_STAT_NXT:
    let i = $i + 1
    goto :FI_STLOOP:
```

```
#####################################################################
#
# Receive Errors screen...
#
#####################################################################

:FI_RCV:
    if $# < 3 then; goto :FI_Rx1:
    let cmdlen = strlen(“$3”)
    let i = strindex(“errors”, “$3”)
    if $i = 1 then ; if $cmdlen >= 3 then ; goto :FI_Rx1:
    goto :FI_HELP:

:FI_Rx1:
    let i = 1

#
# check for circuit name to match on...
#
    if $# <= 3 then ; goto :FI_RCV_NM:
    setenv cmd “$4”
    let len = strlen(cmd)
    if $len < 3 then ; goto :FI_HELP_SHOW:
    let j = strindex(“circuit”, cmd)
    if $j != 1 then ; goto :FI_HELP_SHOW:
    if $# != 5 then ; goto :FI_HELP_SHOW:
    setenv matchcct “$5”

:FI_RCV_NM:

setenv nextflag “”
setenv pattern  “”
setenv nextid   “*”

array    attr wfFDDICct wfFDDISlot wfFDDINode wfFDDICrcErrRx
array -a attr wfFDDIOverrunRx wfFDDIInvalidFrameStatusRx
array -a attr wfFDDIMacErrRx wfFDDIRxOversizedFrames

enum 1  Cct Slot Node CrcErrRx OverrunRx InvalidFrame MacErrRx i
enum $i RxOversized

echo
echo “FDDI Module Receive Errors:”
echo “---------------------------”
echo

printf “%-4.4s %-4.4s %-8.8s %-10.10s %-10.10s %-10.10s %-10.10s\n” \
       “”  “”  “”  “CRC”  “Overrun”  “Invalid”  “Frames”
```

```
printf “%-4.4s %-4.4s %-8.8s %-10.10s %-10.10s %-10.10s %-10.10s\n” \
       “Slot”  “Conn”  “Circuit”  “Errors” “ “Errors” “Frames” “Too \
       Long”
printf “%-4.4s %-4.4s %-8.8s %-10.10s %-10.10s %-10.10s %-10.10s\n” \
       “----”  “----”  “--------”  “----------”  “----------” \
       “----------” “----------”

    let i = 0

:FI_RxLOOP:
    mibget -n wfFddiEntry attr $nextid  value nextid
    if “$nextid“ = “END” then; let i = $i + 1; goto :FOOTER_AND_END:

    let cctnum = ${value[$Cct]}
    gosub :GETCCTNAME:

    if “$matchcct“ = “” then ; goto :FI_PRNTR:
    if -ic “$matchcct“ != “$cctname” then ; goto :FI_RCV_NXT:
    let match_entries = $match_entries + 1

:FI_PRNTR:
    printf “%4d %4d “ ${value[$Slot]} \
    ${value[$Node]}

    printf “%-8.8s %10u %10u ” “$cctname“ ${value[$CrcErrRx]} \
        ${value[$OverrunRx]}
    let invalid = ${value[$InvalidFrame]} + ${value[$MacErrRx]}
    gosub :MAC_EXTRACT:

    let invalid = $invalid + $mac_sum
    printf “%10u %10u ” $invalid ${value[$RxOversized]}
    echo
    cutenv -c9-15 name2  “$cctname“
    if ${name2[0]} = 1 then; \
        printf “%4s %4s *%-7.7s\n” ““ “” “${name2[1]}“

:FI_RCV_NXT:
    let i = $i + 1
    goto :FI_RxLOOP:
#
# Gosub to extract errors in the MAC object from the octet strings...
# ...returns sum in “mac_sum”
#
:MAC_EXTRACT:
    setenv mac_sum 0
    if $?(wfFddiMacEntry.wfFddiMacErrorCts.${nextid}) = 0 then ; \
        return
```

```
    array  mac_attr wfFddiMacErrorCts/HEX_BYTES
wfFddiMacLostCts/HEX_BYTES
    enum 1 errorcts lostcts
    mibget wfFddiMacEntry mac_attr $nextid mac_value mac_next
    cutenv -d. -f1- mac_err  ${mac_value[$errorcts]}
    cutenv -d. -f1- mac_lost ${mac_value[$lostcts]}

    let mac_sum = $mac_sum + 0x${mac_err[8]}
    let mac_sum = $mac_sum + (256 * 0x${mac_err[7]})
    let mac_sum = $mac_sum + (65536 * 0x${mac_err[6]})
    let mac_sum = $mac_sum + (16777216 * 0x${mac_err[5]})

    let mac_sum = $mac_sum + 0x${mac_lost[8]}
    let mac_sum = $mac_sum + (256 * 0x${mac_lost[7]})
    let mac_sum = $mac_sum + (65536 * 0x${mac_lost[6]})
    let mac_sum = $mac_sum + (16777216 * 0x${mac_lost[5]})
    return

######################################################################
#
# Transmit Errors screen...
#
######################################################################

:FI_XMIT:
    if $# < 3 then; goto :FI_Tx1:
    let cmdlen = strlen("$3")
    let i = strindex("errors", "$3")
    if $i = 1 then ; if $cmdlen >= 3 then ; goto :FI_Tx1:
    goto :FI_HELP:

:FI_Tx1:

#
# check for circuit name to match on...
#
    if $# <= 3 then ; goto :FI_XMIT_NM:
    setenv cmd "$4"
    let len = strlen(cmd)
    if $len < 3 then ; goto :FI_HELP_SHOW:
    let j = strindex("circuit", cmd)
    if $j != 1 then ; goto :FI_HELP_SHOW:
    if $# != 5 then ; goto :FI_HELP_SHOW:
    setenv matchcct "$5"
:FI_XMIT_NM:

setenv nextflag ""
setenv pattern  ""
```

```
setenv nextid   "*"
array attr wfFDDICct wfFDDISlot wfFDDINode  wfFDDIAbortTx \
wfFDDIUnderrunTx
enum 1 Cct Slot Node AbortTx UnderrunTx

echo
echo "FDDI Module Transmit Errors:"
echo "----------------------------"
echo

printf "%-4.4s %-4.4s %-8.8s %-10.10s %-10.10s\n" \
       "    " "    " "       " "  Aborted " " Underrun "
printf "%-4.4s %-4.4s %-8.8s %-10.10s %-10.10s\n" \
       "Slot" "Conn" "Circuit" "  Frames  " "  Errors  "
printf "%-4.4s %-4.4s %-8.8s %-10.10s %-10.10s\n" \
       "----" "----" "--------" "----------" "----------"

    let i = 0

:FI_TxLOOP:
    on error :FI_XMIT_NXT:
    mibget -n wfFddiEntry attr $nextid  value nextid
    if "$nextid" = "END" then; let i = $i + 1; goto :FOOTER_AND_END:

    let cctnum = ${value[$Cct]}
    gosub :GETCCTNAME:

    if "$matchcct" = "" then ; goto :FI_PRNTT:
    if -ic "$matchcct" != "$cctname" then ; goto :FI_XMIT_NXT:
    let match_entries = $match_entries + 1

:FI_PRNTT:
    printf "%4d %4d %-8.8s %10u %10u\n" ${value[$Slot]} ${value[$Node]} \
           "$cctname" ${value[$AbortTx]} ${value[$UnderrunTx]}

    cutenv -c9-15 name2  "$cctname"
    if ${name2[0]} = 1 then; \
        printf "%4s %4s *%-7.7s\n" "" "" "${name2[1]}"

:FI_XMIT_NXT:
    let i = $i + 1
    goto :FI_TxLOOP:

#####################################################################
#
# System Errors screen...
#
#####################################################################
```

```
:FI_SYSTEM:
    if $# < 3 then; goto :FI_Sy1:
    let cmdlen = strlen("$3")
    let i = strindex("errors", "$3")
    if $i = 1 then ; if $cmdlen >= 3 then ; goto :FI_Sy1:
    goto :FI_HELP:

:FI_Sy1:

#
# check for circuit name to match on...
#
    if $# <= 3 then ; goto :FI_SYS_NM:
    setenv cmd "$4"
    let len = strlen(cmd)
    if $len < 3 then ; goto :FI_HELP_SHOW:
    let j = strindex("circuit", cmd)
    if $j != 1 then ; goto :FI_HELP_SHOW:
    if $# != 5 then ; goto :FI_HELP_SHOW:
    setenv matchcct "$5"

:FI_SYS_NM:
 setenv nextflag ""
 setenv pattern  ""
 setenv nextid   "*"
 array attr wfFDDICct wfFDDISlot wfFDDINode  wfFDDIParityErrRx \
       wfFDDIParityErrTx
 array -a attr wfFDDIRingErrRx wfFDDIRingErrTx wfFDDISmtRingErrRx \
       wfFDDIPortOpErr
 array -a attr wfFDDIInternOpErr wfFDDIHostErr

 enum  1 Cct Slot Node  ParityErrRx ParityErrTx i
 enum $i RingErrRx RingErrTx SmtRingErrRx PortOpErr InternOpErr HostErr

 echo
 echo "FDDI Module System Errors:"
 echo "--------------------------"
 echo

 printf "%-4.4s %-4.4s %-8.8s %-10.10s %-10.10s %-10.10s %-10.10s \
        %-10.10s\n" "" "" "" "" "" "" "   Port" " Internal" ""
 printf "%-4.4s %-4.4s %-8.8s %-10.10s %-10.10s %-10.10s %-10.10s %-
        10.10s\n" "" "" "" "  Parity" "   Ring" "Operation" "Operation"
        "   Host"
 printf "%-4.4s %-4.4s %-8.8s %-10.10s %-10.10s %-10.10s %-10.10s \
        %-10.10s\n" "Slot" "Conn" "Circuit" "  Errors" "  Errors" \
       "  Errors" "  Errors" "  Errors"
```

```
 printf "%-4.4s %-4.4s %-8.8s %-10.10s %-10.10s %-10.10s %-10.10s \
        %-10.10s\n" "----" "----" "--------" "----------" \
       "----------" "----------" "----------" "----------"
    let i = 0

:FI_SyLOOP:
    mibget -n wfFddiEntry attr $nextid  value nextid
    if "$nextid" = "END" then; let i = $i + 1; goto :FOOTER_AND_END:

    let cctnum = ${value[$Cct]}
    gosub :GETCCTNAME:

    if "$matchcct" = "" then ; goto :FI_PRNTSE:
    if -ic "$matchcct" != "$cctname" then ; goto :FI_SYS_NXT:
    let match_entries = $match_entries + 1

:FI_PRNTSE:

    let parity = ${value[$ParityErrRx]}
    let parity = $parity + ${value[$ParityErrTx]}

    let ring = ${value[$RingErrRx]}
    let ring = $ring + ${value[$RingErrTx]}
    let ring = $ring + ${value[$SmtRingErrRx]}

    printf "%4d %4d %-8.8s " ${value[$Slot]} ${value[$Node]} "$cctname"
    printf "%10u %10u %10u %10u %10u " $parity $ring \
           ${value[$PortOpErr]} ${value[$InternOpErr]} \
           ${value[$HostErr]}

    echo
    cutenv -c9-15 name2  "$cctname"
    if ${name2[0]} = 1 then; \
       printf "%4s %4s *%-7.7s\n" "" "" "${name2[1]}"

:FI_SYS_NXT:
    let i = $i + 1
    goto :FI_SyLOOP:
#####################################################################
#
# SMT screen...
#
#####################################################################

:FI_SMT:
    let i = 1

#
```

```
# check for circuit name to match on...
#
    if $# = 2 then ; goto :FI_SMT_NM:
    setenv cmd "$3"
    let len = strlen(cmd)
    if $len < 3 then ; goto :FI_HELP_SHOW:
    let j = strindex("circuit", cmd)
    if $j != 1 then ; goto :FI_HELP_SHOW:
    if $# != 4 then ; goto :FI_HELP_SHOW:
    setenv matchcct "$4"

:FI_SMT_NM:
echo
echo "FDDI Modules SMT Parameters:"
echo "----------------------------"
echo
printf "%-4.4s %-4.4s %-8.8s %-31.31s %-8.8s %-8.8s %-6.6s\n" \
       "" "" "" " Connection Policy (R = Reject)" "" "" ""
printf "%-4.4s %-4.4s %-8.8s %-31.31s %-8.8s %-8.8s %-6.6s\n" \
       "" "" "" "- - - - - - - - - - - - - - - -" "" "" ""
printf "%18.18s %-31.31s %-8.8s %-8.8s %-6.6s\n" \
       "Local:" "M M M M S S S S B B B B A A A A" "" "" ""
printf "%18.18s %-31.31s %-8.8s %-8.8s %-6.6s\n" \
       "Neighbor:" "M S B A M S B A M S B A M S B A" "" "" "  T"
printf "%-4.4s %-4.4s %-8.8s %-31.31s %-8.8s %-8.8s %-6.6s\n" \
       "" "" "" "| | | | | | | | | | | | | | | |" "   ECM" " Connect" \
       "Notify"
printf "%-4.4s %-4.4s %-8.8s %-31.31s %-8.8s %-8.8s %-6.6s\n" \
       "Slot" "Conn" "Circuit" "| | | | | | | | | | | | | | | |" " \
       State" " State" "(secs)"
printf "%-4.4s %-4.4s %-8.8s %-31.31s %-8.8s %-8.8s %-6.6s\n" \
        "----" "----" "--------" "- - - - - - - - - - - - - - - -" \
       "--------" "--------" "------"

    instenv list_ wfFddiSmtEntry

:FI_SMT_LP:
    on error :FI_SMT_NXT:
    if $i > $list_0 then; goto :FOOTER_AND_END:

    let cctnum = $(wfFddiEntry.wfFDDICct.${list_[$i]})
    gosub :GETCCTNAME:

    if "$matchcct" = "" then ; goto :FI_PRNTSMT:
    if -ic "$matchcct" != "$cctname" then ; goto :FI_SMT_NXT:
    let match_entries = $match_entries + 1

:FI_PRNTSMT:
```

```
    printf "%4d %4d " $(wfFddiEntry.wfFDDISlot.${list_[$i]}) \
    $(wfFddiEntry.wfFDDINode.${list_[$i]})

    printf "%-8.8s " "$cctname"
#
# Print the connection policy matrix...
#
    let bit = 12
    setenv flags ""
    setenv bit0x0 ". . . . "
    setenv bit0x1 ". . . R "
    setenv bit0x2 ". . R . "
    setenv bit0x3 ". . R R "
    setenv bit0x4 ". R . . "
    setenv bit0x5 ". R . R "
    setenv bit0x6 ". R R . "
    setenv bit0x7 ". R R R "
    setenv bit0x8 "R . . . "
    setenv bit0x9 "R . . R "
    setenv bit0xA "R . R . "
    setenv bit0xB "R . R R "
    setenv bit0xC "R R . . "
    setenv bit0xD "R R . R "
    setenv bit0xE "R R R . "
    setenv bit0xF "R R R R "
    let policy = $(wfFddiEntry.wfFDDISmtConnectionPolicy.${list_[$i]})

:FI_BIT_LP:
    let -h result = ($policy & (0xF << $bit)) >> $bit
    setenv flags "${flags}${bit[$result]}"

:FI_NEXT_BIT:
    let bit = $bit - 4
    if $bit >= 0 then ; goto :FI_BIT_LP:

# Note field size is 32 because flags contains the trailing space.
    printf "%32s" "$flags"

    let state = $(wfFddiSmtEntry.wfFddiSmtEcmState.${list_[$i]})
    if $state = 1 then ; printf "%-8s " "ByPassed"
    if $state = 2 then ; printf "%-8s " "In"
    if $state = 3 then ; printf "%-8s " "Trace"
    if $state = 4 then ; printf "%-8s " "Leave"
    if $state = 5 then ; printf "%-8s " "Pathtest"
    if $state = 6 then ; printf "%-8s " "Insert"
    if $state = 7 then ; printf "%-8s " "By_Check"
    if $state = 8 then ; printf "%-8s " "Deinsert"
```

```
    let state = $(wfFddiSmtEntry.wfFddiSmtCfState.${list_[$i]})
    if $state = 1  then ; printf "%-8s " "Isolated"
    if $state = 2  then ; printf "%-8s " "Unknown"
    if $state = 3  then ; printf "%-8s " "Wrap A"
    if $state = 4  then ; printf "%-8s " "Wrap B"
    if $state = 5  then ; printf "%-8s " "Wrap AB"
    if $state = 6  then ; printf "%-8s " "Thru"
    if $state = 7  then ; printf "%-8s " "Local A"
    if $state = 8  then ; printf "%-8s " "Local B"
    if $state = 9  then ; printf "%-8s " "Local AB"
    if $state = 10 then ; printf "%-8s " "Unknown"
    if $state = 11 then ; printf "%-8s " "C Wrap A"
    if $state = 12 then ; printf "%-8s " "C Wrap B"
    if $state = 13 then ; printf "%-8s " "Unknown"

    printf "%6d " $(wfFddiEntry.wfFDDISmtTNotify.${list_[$i]})
#
# Next line...
#
    echo
    cutenv -c9-15 name2  "$cctname"
    if ${name2[0]} = 1 then; \
       printf "%4s %4s *%-7.7s\n" "" "" "${name2[1]}"

:FI_SMT_NXT:
    let i = $i + 1
    goto :FI_SMT_LP:
```

```
#####################################################################
#
# MAC screen...
#
#####################################################################

:FI_MAC:
    let i = 1

#
# check for circuit name to match on...
#
    if $# = 2 then ; goto :FI_MAC_NM:
    setenv cmd "$3"
    let len = strlen(cmd)
    if $len < 3 then ; goto :FI_HELP_SHOW:
    let j = strindex("circuit", cmd)
    if $j != 1 then ; goto :FI_HELP_SHOW:
    if $# != 4 then ; goto :FI_HELP_SHOW:
    setenv matchcct "$4"

:FI_MAC_NM:

echo
echo "FDDI Modules MAC Parameters:"
echo "----------------------------"
echo

printf "%-4.4s %-4.4s %-8.8s %-17.17s %-17.17s %-4.4s %-9.9s \
       %-10.10s\n" "" "" "" "    Upstream" "   Downstream" "TNeg" \
       "Ring Mgmt" " Ring Op"
printf "%-4.4s %-4.4s %-8.8s %-17.17s %-17.17s %-4.4s %-9.9s \
       %-10.10s\n" "Slot" "Conn" "Circuit" "    Neighbor" \
       "    Neighbor" "(ms)" "  State" "  Count"
printf "%-4.4s %-4.4s %-8.8s %-17.17s %-17.17s %-4.4s %-9.9s  \
       %-10.10s\n" "----" "----" "--------" "-----------------" \
       "-----------------" "----" "---------" "----------"

    instenv list_ wfFddiMacEntry
```

```
:FI_MAC_LP:
    on error :FI_MAC_NXT:
    if $i > $list_0 then; goto :FOOTER_AND_END:

    let cctnum = $(wfFddiEntry.wfFDDICct.${list_[$i]})
    gosub :GETCCTNAME:

    if "$matchcct" = "" then ; goto :FI_PRNTMAC:
    if -ic "$matchcct" != "$cctname" then ; goto :FI_MAC_NXT:
    let match_entries = $match_entries + 1

:FI_PRNTMAC:
    printf "%4d %4d " $(wfFddiMacEntry.wfFddiMacSlot.${list_[$i]}) \
    $(wfFddiMacEntry.wfFddiMacNode.${list_[$i]})

    printf "%-8.8s " "$cctname"

#
# Print Upstream Neighbor's MAC address...
#
    octetfmt mac MAC_ADDRESS
wfFddiMacEntry.wfFddiMacUpstreamNbr.${list_[$i]}
    printf "%-17s " "$mac"
#
# Print Downstream Neighbor's MAC address...
#
    octetfmt mac MAC_ADDRESS
wfFddiMacEntry.wfFddiMacDownstreamNbr.${list_[$i]}
    printf "%-17s " "$mac"
#
# Print remainder of line...
#

    let tneg =  $(wfFddiMacEntry.wfFddiMacTNeg.${list_[$i]}) / 1000000
    printf "%4d " $tneg

    let state = $(wfFddiMacEntry.wfFddiMacRmtState.${list_[$i]})
    if $state =   1 then ; printf "%-9s " "Isolated"
    if $state =   2 then ; printf "%-9s " "NonOp"
    if $state =   4 then ; printf "%-9s " "RingOp"
    if $state =   8 then ; printf "%-9s " "Detect"
    if $state =  16 then ; printf "%-9s " "NonOpDup"
    if $state =  32 then ; printf "%-9s " "RingOpDup"
    if $state =  64 then ; printf "%-9s " "Directed"
    if $state = 128 then ; printf "%-9s " "Trace"
```

```
    printf "%10u\n" $(wfFddiMacExtEntry.wfFddiMacRingOpCts.${list_[$i]})

    cutenv -c9-15 name2  "$cctname"
    if ${name2[0]} = 1 then; \
        printf "%4s %4s *%-7.7s\n" "" "" "${name2[1]}"

#
# Next line...
#

:FI_MAC_NXT:
    let i = $i + 1
    goto :FI_MAC_LP:

######################################################################
#
# Port screen...
#
######################################################################
:FI_PORT:
    let i = 1

echo
echo "FDDI Modules Port Parameters:"
echo "-----------------------------"
echo

printf "%-4.4s %-4.4s %-5.5s %-9.9s %-9.9s %-10.10s %-10.10s \
       %-10.10s\n" ""  ""  ""  "" "" "Link Error" "Elasticity" "   Link"
printf "%-4.4s %-4.4s %-5.5s %-9.9s %-9.9s %-10.10s %-10.10s \
       %-10.10s\n" ""  "" "Local" "Neighbor" " Physical" " Monitor" \
       "  Buffer" "Confidence"
printf "%-4.4s %-4.4s %-5.5s %-9.9s %-9.9s %-10.10s %-10.10s \
       %-10.10s\n" "Slot" "Conn" "Port" "Port Type" "  State" " \
       Count" "  Errors" "  Count"
printf "%-4.4s %-4.4s %-5.5s %-9.9s %-9.9s %-10.10s %-10.10s \
       %-10.10s\n" "----" "----" "-----" "---------" "---------" \
       "----------" "----------" "----------"

    instenv list_ wfFddiPortEntry
```

```
:FI_PORT_LP:
    on error :FI_PORT_NXT:
    if $i > $list_0 then; goto :FOOTER_AND_END:
    printf "%4d %4d " $(wfFddiPortEntry.wfFddiPortSlot.${list_[$i]}) \
    $(wfFddiPortEntry.wfFddiPortNode.${list_[$i]})

    let port = $(wfFddiPortEntry.wfFddiPortPcType.${list_[$i]})
    if $port = 1 then ; printf "%-5s " "A"
    if $port = 2 then ; printf "%-5s " "B"
    if $port = 3 then ; printf "%-5s " "S"
    if $port = 4 then ; printf "%-5s " "M"
    if $port = 5 then ; printf "%-5s " "None"

    let port = $(wfFddiPortEntry.wfFddiPortPcNeighbor.${list_[$i]})
    if $port = 1 then ; printf "%-9s " "A"
    if $port = 2 then ; printf "%-9s " "B"
    if $port = 3 then ; printf "%-9s " "S"
    if $port = 4 then ; printf "%-9s " "M"
    if $port = 5 then ; printf "%-9s " "Unknown"

    let port = $(wfFddiPortEntry.wfFddiPortPcmState.${list_[$i]})
    if $port =  1 then ; printf "%-9s " "Off"
    if $port =  2 then ; printf "%-9s " "Break"
    if $port =  3 then ; printf "%-9s " "Trace"
    if $port =  4 then ; printf "%-9s " "Connect"
    if $port =  5 then ; printf "%-9s " "Next"
    if $port =  6 then ; printf "%-9s " "Signal"
    if $port =  7 then ; printf "%-9s " "Join"
    if $port =  8 then ; printf "%-9s " "Verify"
    if $port =  9 then ; printf "%-9s " "Active"
    if $port = 10 then ; printf "%-9s " "Maint"

    printf "%10u " $(wfFddiPortExtEntry.wfFddiPortLemCts.${list_[$i]})
    printf "%10u "
$(wfFddiPortExtEntry.wfFddiPortEbErrorCts.${list_[$i]})
    printf "%10u "
$(wfFddiPortExtEntry.wfFddiPortLctFailCts.${list_[$i]})
#
# Next line...
#
    echo

:FI_PORT_NXT:
    let i = $i + 1
    goto :FI_PORT_LP:
```

```
#####################################################################
#
#  Sample screen (samples parameters twice, prints deltas)
#
#####################################################################

:FI_SAMP:
    let period = 10
    if $# < 3 then; goto :FI_Samp1:
#
# check for circuit to match without period given
#
    setenv cmd “$3”
    let j = strindex(“circuit”, cmd)
    if $j != 1 then; goto :FI_ITSPERIOD:
    let len = strlen(cmd)
    if $len < 3 then; goto :FI_HELP_SHOW:
    if $# != 4 then; goto :FI_HELP_SHOW:
    setenv matchcct “$4”
    goto :FI_Samp1:

:FI_ITSPERIOD:
    setenv period “$3”
#
# Check “period” parameter for invalid characters, 0 value
#

    if “$ver” < “x7.80” then; goto :FI_SAMP_CKVAL:
    setenv teststr “ABCDEFGHIJKLMNOPQRSTUVWXYZ”
    let invalch = strpbrk(“$3”, “$teststr”)
    if $invalch != 0 then ; goto :FI_SAMP_HELP:

    setenv teststr “abcdefghijklmnopqrstuvwxyz”
    let invalch = strpbrk(“$3”, “$teststr”)
    if $invalch != 0 then ; goto :FI_SAMP_HELP:

    setenv teststr “!@#%^&*()_-=+\|`~[{]}';:/?,<>”
    let invalch = strpbrk(“$3”, “$teststr”)
    if $invalch != 0 then ; goto :FI_SAMP_HELP:
```

```
:FI_SAMP_CKVAL:
    if $period <= 0 then; \
        echo “Period of $period seconds is too low.” ; \
        goto :FI_HELP_SHOW:
#
# Check for circuit to match on...
#
    if $# <= 3 then ; goto :FI_Samp1:
    setenv cmd “$4”
    let len = strlen(cmd)
    if $len < 3 then ; goto :FI_HELP_SHOW:
    let j = strindex(“circuit”, cmd)
    if $j != 1 then ; goto :FI_HELP_SHOW:
    if $# != 5 then ; goto :FI_HELP_SHOW:
    setenv matchcct “$5”
#
#  Take the first sample...
#
:FI_Samp1:
    echo
    printf “FDDI Sampled Data over %3u seconds\n” $period
    echo   “----------------------------------”
    echo

    printf “%4.4s %4.4s %-8.8s %-10.10s %-10.10s %-10.10s %-10.10s\n” \
           “” “” “” “    Rx” “    Tx” “Rx Lack of” “Tx Lack of”
    printf “%4.4s %4.4s %-8.8s %-10.10s %-10.10s %-10.10s %-10.10s\n” \
           “Slot” “Conn” “Circuit” “  Frames” “  Frames” “Resources”\
           “Resources”
    printf “%4.4s %4.4s %-8.8s %10.10s %10.10s %10.10s %10.10s\n” \
    “----” “----” “--------” “----------” “----------” “----------” \
    “----------”

    printf “Taking first sample...”
    instenv list_ wfFddiEntry
    let i = 1

    if $list_0 = 0 then; \
        printf “\r                      \r”; \
        goto :FOOTER_AND_END:
```

```
:FI_SAMP1_LP:
    on error :FI_SAMP1_SKP:
    if $i > $list_0 then; goto :FI_SAMP_WT:

    if "$matchcct" = "" then ; goto :FI_SAMP1_NOW:
    let cctnum = $(wfFddiEntry.wfFDDICct.${list_[$i]})
    gosub :GETCCTNAME:
    if -ic "$matchcct" != "$cctname" then; goto :FI_SAMP1_SKP:

:FI_SAMP1_NOW:
    let match_entries = $match_entries + 1
    let m = $match_entries
    setenv s1instid_$m "${list_[$i]}"
    setenv s1rxfrms_$m "$(wfFddiEntry.wfFDDIFramesRxOk.${list_[$i]})"
    setenv s1txfrms_$m "$(wfFddiEntry.wfFDDIFramesTxOk.${list_[$i]})"
    setenv s1rxlors_$m "$(wfFddiEntry.wfFDDIRingOverrunRx.${list_[$i]})"
    setenv s1txlors_$m "$(wfFddiEntry.wfFDDITxClipFrames.${list_[$i]})"

:FI_SAMP1_SKP:
    let i = $i + 1
    goto :FI_SAMP1_LP:
#
# Wait the proscribed period of time...
#
:FI_SAMP_WT:
    printf "\r                        \r"
    if $match_entries = 0 then; goto :FOOTER_AND_END:
    let s1instid_0 = $match_entries

    printf "Waiting %2u seconds..." $period
    pause  $period
#
# Take the second sample.  Collecting the second sample in its entirety
# before calculating output is preferred because the variance in sample
# times between instances of the MIB object should be reduced.
#
    printf "\r                           \r"
    printf "Taking second sample..."
    instenv list_ wfFddiEntry
    let i = 1
    let match_entries = 0

    if $list_0 = 0 then; \
        printf "\r                         \r"; \
        goto :FOOTER_AND_END:

:FI_SAMP2_LP:
    on error :FI_SAMP2_SKP:
```

```
    if $i > $list_0 then; goto :FI_SAMP_PRT:

    if "$matchcct" = "" then ; goto :FI_SAMP2_NOW:
    let cctnum = $(wfFddiEntry.wfFDDICct.${list_[$i]})
    gosub :GETCCTNAME:
    if -ic "$matchcct" != "$cctname" then; goto :FI_SAMP2_SKP:

:FI_SAMP2_NOW:
    let match_entries = $match_entries + 1
    let m = $match_entries
    setenv s2instid_$m "${list_[$i]}"
    setenv s2rxfrms_$m "$(wfFddiEntry.wfFDDIFramesRxOk.${list_[$i]})"
    setenv s2txfrms_$m "$(wfFddiEntry.wfFDDIFramesTxOk.${list_[$i]})"
    setenv s2rxlors_$m "$(wfFddiEntry.wfFDDIRingOverrunRx.${list_[$i]})"
    setenv s2txlors_$m "$(wfFddiEntry.wfFDDITxClipFrames.${list_[$i]})"

:FI_SAMP2_SKP:
    let i = $i + 1
    goto :FI_SAMP2_LP:

#
# Compare the two samples and generate the output.
#
:FI_SAMP_PRT:

    let table_size = $i
    printf "\r                          \r"
    if $match_entries = 0 then; goto :FOOTER_AND_END:
    let s2instid_0 = $match_entries

    let i = 1
    let j = 1
    let jlow = 1
    let matches = 0
```

```
:FI_SAMP_MATCH:
    on error :FI_NEXT_TRY:
    if ${s1instid_[$i]} != ${s2instid_[$j]} then; goto :FI_NEXT_TRY:
    if $j = $jlow then; \
         if $jlow < $s2instid_0 then; \
            let jlow = $jlow + 1

    let delta_rxfrms = ${s2rxfrms_[$j]} - ${s1rxfrms_[$i]}
    if $delta_rxfrms < 0 then; goto :FI_NEXT_TRY:
    let delta_txfrms = ${s2txfrms_[$j]} - ${s1txfrms_[$i]}
    if $delta_txfrms < 0 then; goto :FI_NEXT_TRY:
    let delta_rxlors = ${s2rxlors_[$j]} - ${s1rxlors_[$i]}
    if $delta_rxlors < 0 then; goto :FI_NEXT_TRY:
    let delta_txlors = ${s2txlors_[$j]} - ${s1txlors_[$i]}
    if $delta_txlors < 0 then; goto :FI_NEXT_TRY:

    let matches = $matches + 1
    let cctnum = $(wfFddiEntry.wfFDDICct.${s1instid_[$i]})
    gosub :GETCCTNAME:
    printf "%4d %4d " $(wfFddiEntry.wfFDDISlot.${s1instid_[$i]}) \
           $(wfFddiEntry.wfFDDINode.${s1instid_[$i]})
    printf "%-8.8s " "$cctname"

    printf "%10u %10u %10u %10u\n" $delta_rxfrms $delta_txfrms \
                                   $delta_rxlors $delta_txlors

    cutenv -c9-15 name2  "$cctname"
    if ${name2[0]} = 1 then; \
       printf "%4s %4s *%-7.7s\n" "" "" "${name2[1]}"

:FI_NEXT_TRY:
    let j = $j + 1
    if $j <= $s2instid_0 then; goto :FI_SAMP_MATCH:

    let i = $i + 1
    if $i <= $s1instid_0 then; \
       let j = $jlow; \
       goto :FI_SAMP_MATCH:

    let i = $table_size
    goto :FOOTER_AND_END:

:FI_SAMP_HELP:
    cutenv -c $invalch ugly_ "$3"
    echo "Inappropriate character '$ugly_1' in parameter;"
    echo "   only digits allowed."
    echo
    goto :FI_HELP_SHOW:
```

```
#####################################################################
#
# Hardware Filter screen...
#
#####################################################################

:FI_HWF:
setenv RUN_SILENT 1
on error :FI_HANDLER:

run hwfilter show fddi
goto :FI_END:

:FI_HANDLER:
echo “Script hwfilter.bat not found.”
echo
goto :FI_END:

#####################################################################
#
#  Parse for “enable” command...
#
#####################################################################

:FI_ENBL:
    if $# = 1 then ; goto :FI_HELP_ENBL:
    setenv cmd “$2”
    let len = strlen(cmd)
    if $len < 3 then ; goto :FI_HELP_ENBL:

    let i = strindex(“circuit”, cmd)
    if $i = 1 then ; goto :FI_ENBLC:

    let i = strindex(“connector”, cmd)
    if $i = 1 then ; goto :FI_ENBLI:

:FI_HELP_ENBL:
    echo “FDDI Enable Command Help”
    echo “------------------------”
    echo
    echo “enable fddi <option>”
    echo “   Where option is one of the following:”
    echo “   ?”
    echo “   circuit   <circuit name>”
    echo “   connector <slot.connector>”
    goto :FI_END:
```

```
#
# Enable circuit...
#

:FI_ENBLC:
    if $# != 3 then ; goto :FI_HELP_ENBL:

    instenv list_ wfFddiEntry
    if $list_0 = 0 then; goto :FI_ENBLNF:

    setenv matchcct "$3"
    if "$matchcct" = "" then ; goto :FI_END:

    let i = 1

# Search for circuit with matching name

:FI_ENBLLP:
    on error :FI_ENBLNXT:
    if $i > $list_0 then; goto :FI_ENBLNF:

    let cctnum = $(wfFddiEntry.wfFDDICct.${list_[$i]})
    gosub :GETCCTNAME:
    if -ic "$matchcct" != "$cctname" then ; goto :FI_ENBLNXT:

# Found it!

    set wfFddiEntry.wfFDDIEnable.${list_[$i]} 1
    echo "FDDI circuit $cctname enabled."
    commit
    goto :FI_END:

:FI_ENBLNXT:
    let i = $i + 1
    goto :FI_ENBLLP:

:FI_ENBLNF:
    echo "FDDI circuit $3 not found."
    goto :FI_END:

#
# Enable connector...
#
```

```
:FI_ENBLI:
    if $# != 3 then ; goto :FI_HELP_ENBL:

# Sanity check <slot.connector> parameter...

    if "$ver" < "x7.80" then; goto :SKIP_STRPBRK1:
    setenv teststr "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    let invalch = strpbrk("$3", "$teststr")
    if $invalch != 0 then ; goto :FIEN_MSG1_HELP:

    setenv teststr "abcdefghijklmnopqrstuvwxyz"
    let invalch = strpbrk("$3", "$teststr")
    if $invalch != 0 then ; goto :FIEN_MSG1_HELP:

    setenv teststr "!@#%^&*()_-=+\|`~[{]}';:/?,<>"
    let invalch = strpbrk("$3", "$teststr")
    if $invalch != 0 then ; goto :FIEN_MSG1_HELP:

:SKIP_STRPBRK1:
    let firstdot = strindex("$3", ".")
    if $firstdot = 0 then ; goto :FIEN_MSG2_HELP:
    let lastdot = strrindex("$3", ".")
    if $firstdot != $lastdot then ; goto :FIEN_MSG3_HELP:

    if $?(wfFddiEntry.wfFDDIEnable.$3) = 0 then ; goto :FI_ENBLIE:

    set wfFddiEntry.wfFDDIEnable.$3 1
    echo "FDDI connector $3 enabled."
    commit
    goto :FI_END:

:FI_ENBLIE:
    echo "FDDI connector $3 not found."
    goto :FI_END:

:FIEN_MSG1_HELP:
    gosub :ERR_MSG1:
    goto :FI_HELP_ENBL:

:FIEN_MSG2_HELP:
    gosub :ERR_MSG2:
    goto :FI_HELP_ENBL:

:FIEN_MSG3_HELP:
    gosub :ERR_MSG3:
    goto :FI_HELP_ENBL:
```

```
#
# Gosubs shared by enable and disable...
#

:ERR_MSG1:
    cutenv -c $invalch ugly_ “$3”
    echo “Inappropriate character '$ugly_1' in parameter;”
    echo “   only digits and '.' allowed.”
    echo
    return

:ERR_MSG2:
    echo “Parameter must contain a '.' character.”
    echo
    return

:ERR_MSG3:
    echo “Too many '.' characters in parameter; enter one only.”
    echo
    return
#####################################################################
#
#  Parse for “disable” command...
#
#####################################################################

:FI_DSBL:
    if $# = 1 then ; goto :FI_HELP_DSBL:
    setenv cmd “$2”
    let len = strlen(cmd)
    if $len < 3 then ; goto :FI_HELP_DSBL:

    let i = strindex(“circuit”, cmd)
    if $i = 1 then ; goto :FI_DSBLC:

    let i = strindex(“connector”, cmd)
    if $i = 1 then ; goto :FI_DSBLI:

:FI_HELP_DSBL:
    echo “FDDI Disable Command Help”
    echo “-------------------------”
    echo
    echo “disable fddi <option>”
    echo “   Where option is one of the following:”
    echo “   ?”
    echo “   circuit   <circuit name>”
    echo “   connector <slot.connector>”
    goto :FI_END:
```

```
#
# Disable circuit...
#

:FI_DSBLC:
    if $# != 3 then ; goto :FI_HELP_DSBL:

    instenv list_ wfFddiEntry
    if $list_0 = 0 then; goto :FI_DSBLNF:

    setenv matchcct "$3"
    if "$matchcct" = "" then ; goto :FI_END:

    let i = 1

# Search for circuit with matching name

:FI_DSBLLP:
    on error :FI_DSBLNXT:
    if $i > $list_0 then; goto :FI_DSBLNF:

    let cctnum = $(wfFddiEntry.wfFDDICct.${list_[$i]})
    gosub :GETCCTNAME:
    if -ic "$matchcct" != "$cctname" then ; goto :FI_DSBLNXT:

# Found it!

    set wfFddiEntry.wfFDDIEnable.${list_[$i]} 2
    echo "FDDI circuit $cctname disabled."
    commit
    goto :FI_END:

:FI_DSBLNXT:
    let i = $i + 1
    goto :FI_DSBLLP:

:FI_DSBLNF:
    echo "FDDI circuit $3 not found."
    goto :FI_END:

#
# Disable connector...
#
```

```
:FI_DSBLI:
    if $# != 3 then ; goto :FI_HELP_DSBL:

# Sanity check <slot.connector> parameter...

    if "$ver" < "x7.80" then; goto :SKIP_STRPBRK2:
    setenv teststr "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    let invalch = strpbrk("$3", "$teststr")
    if $invalch != 0 then ; goto :FIDS_MSG1_HELP:

    setenv teststr "abcdefghijklmnopqrstuvwxyz"
    let invalch = strpbrk("$3", "$teststr")
    if $invalch != 0 then ; goto :FIDS_MSG1_HELP:

    setenv teststr "!@#%^&*()_-=+\|`~[{]}';:/?,<>"
    let invalch = strpbrk("$3", "$teststr")
    if $invalch != 0 then ; goto :FIDS_MSG1_HELP:

:SKIP_STRPBRK2:
    let firstdot = strindex("$3", ".")
    if $firstdot = 0 then ; goto :FIDS_MSG2_HELP:
    let lastdot = strrindex("$3", ".")
    if $firstdot != $lastdot then ; goto :FIDS_MSG3_HELP:

    if $?(wfFddiEntry.wfFDDIEnable.$3) = 0 then ; goto :FI_DSBLIE:

    set wfFddiEntry.wfFDDIEnable.$3 2
    echo "FDDI connector $3 disabled."
    commit
    goto :FI_END:

:FI_DSBLIE:
    echo "FDDI connector $3 not found."
    goto :FI_END:

:FIDS_MSG1_HELP:
    gosub :ERR_MSG1:
    goto :FI_HELP_DSBL:

:FIDS_MSG2_HELP:
    gosub :ERR_MSG2:
    goto :FI_HELP_DSBL:

:FIDS_MSG3_HELP:
    gosub :ERR_MSG3:
    goto :FI_HELP_DSBL:
```

```
#####################################################################
#
# Version command
#
#####################################################################

:FI_VER:
    echo “fddi.bat Version: 1.11 Date: 10/31/94.”
    goto :FI_END:

#
# Circuit Name fetching Gosub...
#
:GETCCTNAME:
    if $?(wfCircuitNameEntry.wfCircuitName.$cctnum) = 0 then ; \
        setenv cctname “” ; \
        return
    setenv cctname “$(wfCircuitNameEntry.wfCircuitName.$cctnum)”
    return

#
# Branch point to print footer of table, then fall through to end of
# script.
#
:FOOTER_AND_END:

    echo
    let tbl_entries = $i - 1
    if “$matchcct” = “” then ; goto :FAE_SKIP:
    if $match_entries = 1 then ; \
        printf “Found %5d match out of ” $match_entries; goto :FAE_SKIP:
     printf “Found %5d matches out of ” $match_entries

:FAE_SKIP:
     if $tbl_entries = 1 then ; \
        printf “%5d entry in table.\n” $tbl_entries ; \
        goto :FAE_DONE:
    printf “%5d entries in table.\n” $tbl_entries
:FAE_DONE:

#
# Exit branch point...
#
:FI_END:
```

Index

G

I

L

M

O

P

R

S

T

U

V

W

X